Model-Driven Game Development
Addressing Architectural Diversity
and Game Engine-Integration

Meng Zhu

Doctoral Thesis
Submitted for the Partial Fulfillment of the Requirements for the Degree of
philosophiae doctor

Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering
Norwegian University of Science and Technology
March 2014
To my father Ruixing Zhu who passed away during the Ph.D. research, but his love will encourage me forever.
Abstract

Model-Driven Game Development (MDGD) is an emerging research field, which uses models to specify some or all of the game elements that traditionally had been manually coded. The PhD thesis presents research in the MDGD domain, which is intended to push MDGD further towards industrialization by reducing two gaps: 1) The integration of MDGD tools and game engines, and 2) The support for a diversity of game architectures. These gaps have been identified through a literature review of existing MDGD approaches, which is also documented in the thesis.

To reduce the first gap, the model-driven approach Engine Cooperative Game Modeling (ECGM) has been proposed, which uses a run-time game engine as the base for building a domain framework, and engine tools together with MDGD tools for creating game code and data. The code generator generates both game code and level data from game models, making the game models operable in the engine tools. ECGM has been evaluated through being instantiated with a Domain Specific Language (DSL), Reactive AI Language (RAIL) tools, and the commercial game engine Torque 2D. The DSL and engine were used to develop a prototype game, whose evaluation showed that ECGM can significantly improve the productivity and enable an efficient workflow.

To reduce the second gap, the game architecture framework Game Worlds Graph (GWG) has been proposed. GWG has then been used as the conceptual base for a MDGD approach supporting a diversity of game architectures. The GWG-based MDGD approach employs the Global World, Local World, and Connector Type concepts in modeling languages, adding the architectural information to the modeled game elements, with which code for different architectures can be generated. To evaluate the approach, a DSL and its tool-chain were created following the approach, and a prototype game supporting three game architectures were developed. The results of the prototyping showed that the MDGD approach and the DSL resulted in significant improvement in productivity and workflow efficiency.

Apart from providing a major research contribution bridging the two gaps described above, the GWG framework itself is a valuable research contribution. The framework can be used for 1) analyzing and classifying existing game architectures, which was proven through a systematic review of 40 game architectures, 2) exploring future architectures resulting in the design of a new game architecture, and 3) aiding MDGD. The RAIL and an extended version of RAIL are also themselves valuable research contributions. The RAIL is a DSL for modeling high-level AI in action/adventure games, and the extended RAIL adds network architecture support to RAIL. Both DSLs are supported by their tool-chains including a model-editor, a semantics validator, and a code generator. These two versions of RAIL can be reused in related MDGD projects or be used as the reference DSLs.
Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) for partial fulfilment of the requirements for the degree of Philosophiae Doctor.

This doctoral work has been performed at the Department of Computer and Information Science, NTNU, Trondheim, Norway under the supervision of Professor Alf Inge Wang as the main supervisor and Associate Professor Hallvard Trætteberg as the co-supervisor.

The doctoral work is financed by the Faculty of Information Technology, Mathematics, and Electrical Engineering (IME), NTNU.
Acknowledgements

The work documented in the thesis was done with the help of many people, and I am grateful to all of them.

Firstly, I would like to express my sincere gratitude to my supervisor Prof. Alf Inge Wang for the continuous support of my Ph.D study with his patience, motivation, and immense knowledge. His guidance is very helpful throughout the research and the writing of this thesis. His great personality makes our cooperation and communication effective, fruitful and memorable. He always has a positive attitude to my ideas, and is always considerate. I have to say it is my luck to have such a super supervisor.

Secondly, I must thank my co-supervisor Hallvard Trætteburg for his insights and inspirations on every workshop. He enlightened me the concepts of model-driven development thus opened a door to the research for me. He helped me a lot on both work and life, especially when Alf Inge was not in Trondheim. I will never forget the day when he showed me around the city, the dinners with his family and much more…

Besides my supervisors, I would like to thank the rest of my thesis committee: Prof. Walt Scacchi, Prof. Kendra Cooper and Prof. Konstantinos Chorianopoulos for reviewing and commenting on my thesis, as well as organizing the defense.

I would like to thank the colleagues in IDI, who have providing valuable input to my research plan and preliminary results. I would especially thank Prof. Reidar Conradi, Prof. Monica Divitini, and Prof. John Krogstie for their useful courses that directly or indirectly helped my research.

My sincere thanks also goes to my friends and colleagues Dr. Bian Wu, Dr. Shang Gao, Feng Luan and Dr. Chengzhi Liu, whose inspirations are valuable to my research and our friendship makes the time after work fun and memorable.

Last but not the least, I would like to thank my mother for supporting me spiritually throughout writing this thesis. I would also thank my daughter Anne for making me a father which is so amazing, and I must thank her mother Dr. Hong Guo for all the happy days and all the difficult days we went through together. When we came to Norway seven years ago we were not confident of finishing the Ph.D. but we finally did it! I would also thank my father who passed away during my research, but his love is the source of my energy for finishing the thesis and more.
## Contents

1 Introduction ......................................................................................................................... 1  
   1.1 Developing A Game Is Not A Game ................................................................. 1  
   1.2 Why Model Games ..................................................................................... 3  
   1.3 Research Questions .................................................................................... 5  
   1.4 Research Methodology and Design .......................................................... 9  
      1.4.1 Research Methodology ...................................................................... 9  
      1.4.2 Research Design ............................................................................... 10  
   1.6 Thesis Structure ......................................................................................... 12  
2 Towards Model-Driven Game Development (MDGD) Industrialization ... 15  
   2.1 Model-Driven Development: Basic Concepts ....................................... 15  
   2.2 An Overview of Exiting MDGD Approaches ........................................ 17  
   2.3 Domain Definitions .................................................................................. 22  
   2.4 What Is A Proper Domain Definition? ...................................................... 25  
   2.5 Domain Frameworks and Target Environment ....................................... 27  
   2.6 Modeling Languages ................................................................................ 28  
      2.6.1 Language Fundamentals ................................................................. 28  
      2.6.2 Model of Computation .................................................................. 31  
   2.6 MDGD Tooling ....................................................................................... 36  
      2.6.1 Tooling of the MDGD Approaches .................................................. 37  
      2.6.2 Challenges with MDGD Tooling ...................................................... 41  
   2.7 Game Architectural Concerns ................................................................. 44  
      2.7.1 The Diversity of Game Architectures .............................................. 44  
      2.7.2 The Proper Abstraction of Game Architectures ............................... 45  
   2.8 Summary ........................................................................................................ 45  
3 Engine- Cooperative Game Modeling (ECGM): A Hybrid Approach  
   Bridging MDGD Tools and Game Engine Tools ................................................. 49  
   3.1 Game Engine: The Past and The Present ................................................... 49  
   3.2 Modern Game Engine Architecture .......................................................... 51  
   3.3 Some Game Engines In Market .................................................................. 54  
      3.3.1 Unreal Engines .............................................................................. 54  
      3.3.2 Unity Engine .................................................................................. 55  
      3.3.3 Torque Engines ............................................................................ 56  
   3.4 Engine-Based Game Development ............................................................. 56  
   3.5 ECGM Framework .................................................................................... 58  
   3.6 Summary ....................................................................................................... 62  
4 Case Study 1: Reactive AI Language (RAIL) and Orcs’ Gold ..................... 65
List of Figures

Figure 1.1 Research Design ........................................................................................................... 10
Figure 2.1 MDD Concepts and Their Relationships ................................................................. 17
Figure 2.2 Domain Scopes of The Reviewed MDGD Approaches ............................................ 25
Figure 2.3 Graphical Concrete Syntax V.S. Textual Concrete Syntax ........................................ 31
Figure 2.4 Screenshot of Two Level Editors From Commercial Game Engines ...................... 34
Figure 2.5 (a) Secret Zone in World 2-1 and (b) Level Transition Tree (Relevant Part) of Super Mario Bros ................................................................................................................. 36
Figure 3.1 General Architecture of Modern Game Engines ...................................................... 52
Figure 3.2 Architecture of Engine-Based Development ............................................................. 57
Figure 3.3 ECGM Architecture ................................................................................................. 60
Figure 3.4 Level Transition DSL Model and Level Design View Of World 1-2 in Super Mario Bros ......................................................................................................................................... 60
Figure 3.5 Level Transition Modeling with ECGM ..................................................................... 62
Figure 4.1 In-game Screen Shots of Mark of Ninja ..................................................................... 65
Figure 4.2 Screen Shots of Pac-Man .......................................................................................... 66
Figure 4.3 RAIL Meta-Model (Simplified) .................................................................................. 68
Figure 4.4 A RAIL Model in Editor ............................................................................................ 70
Figure 4.5 The Tool Chain Architecture For RAIL Modeling with Torque 2D ......................... 74
Figure 4.6 Use AlIPattern in TGB ............................................................................................... 75
Figure 4.7 An In-Game Screenshot of Orc’s Gold ....................................................................... 76
Figure 5.1 Two Examples of Game Worlds Graph .................................................................... 86
Figure 5.2 GWG as an Architectural Taxonomy Framework .................................................... 91
Figure 5.3 Classification with Graph Styles .............................................................................. 93
Figure 5.4 Classification According to GWG Flavors ................................................................. 94
Figure 5.5 The GWG of Strifeshadow Fantasy ........................................................................... 95
Figure 5.6 An Example GWG of VoroGame ............................................................................. 96
Figure 5.7 A GWG of Rokkatan ............................................................................................... 97
Figure 5.8 A Hybrid Remote-Rendering Architecture with 1-n+n GWG Flavor ....................... 99
Figure 6.1 New Super Mario Bros Wii ....................................................................................... 112
Figure 6.2 Rayman Raving Rabbits .......................................................................................... 112
Figure 6.3 Mario Kart ................................................................................................................. 114
Figure 6.4 Ridge Racer .............................................................................................................. 114
Figure 6.5 Real Tournament[150] ............................................................................................. 122
Figure 6.6 Rokkatan[142] .......................................................................................................... 123
Figure 7.1 Screenshot of Some Multi-Player on Shared-Screen Games ................................... 130
Figure 7.2 Multiplayer Online Games ....................................................................................... 131
Figure 7.3 RAIL Model and The Desired C-S Code ................................................................. 132
Figure 7.4 Organize The Game Model with Game Worlds ....................................................... 134
Figure 7.5 Multi-Global World Game Model and The Target Game ...................................... 135
Figure 7.6 Organize Model Elements with GWG for (a) Tree View, (b) Textual and (c) Graphical Concrete Syntaxes ................................................................................................................................. 137
Figure 7.7 Add Connector Type to A Game Model ................................................................... 138
Figure 7.8 Example of Code Generation Process Based on GWG ........................................ 138
Figure 8.1 Add GWG Elements To RAIL .............................................................................. 143
Figure 8.2 An Example of The RAIL Model and Code Generation .......................................... 145
Figure 8.3 Screenshots of *Networked Orc’s Gold*: Client-Server Version.................. 146
Figure 8.4 Screenshots of *Networked Orc’s Gold*: Host-Joiner Version .................. 147
Figure 8.5 Networked Orc’s Gold RAIL Model .......................................................... 147
Figure 8.6 Three Steps Extending RAIL Model to Networked Architecture .......... 149
## List of Tables

Table 1.1 Contributions .................................................................................................................................................. 10  
Table 2.1 MDGD Approaches in Literature .................................................................................................................. 19  
Table 2.2 Modeling Language Fundamentals of the MDGD Approaches ..................................................................... 29  
Table 2.3 Computational Models of Modeling Languages ............................................................................................... 33  
Table 2.4 The Tool Support and The Meta-Tools of the MDGD Approaches ......................................................... 37  
Table 4.1 Comparison of Manual Coding and MDGD in Developing *Orc’s Gold* ................................................. 76  
Table 4.2 Cost of Developing RAIL and Its Tool-Chain ............................................................................................... 77  
Table 5.1 GWG Game Flavor Expression Examples ...................................................................................................... 90  
Table 6.1 Classification of Architectures According to the GWG framework ......................................................... 109  
Table 6.2 Mapping the Terms Describing Game Architectures to the GWG Flavors ................................................ 110  
Table 6.3 GWG Flavors and The Corresponding Game Types ..................................................................................... 127  
Table 8.1 Comparison of Development Investment with Two Methods ........................................................................ 148  
Table 8.2 Cost of Developing Extended RAIL and Its Tool-Chain ................................................................................ 150  
Table 9.1 Contributions, Publications and Planned Publications ................................................................................ 160
1 Introduction

Games are fun to play but difficult to develop. Major challenges connected to game development include changing requirements, cross-disciplinary communications, creative process, tools and workflow, and so forth. Model-Driven Game Development (MDGD) was proposed by the game community to address the issues. The Ph.D. thesis presents my research in the MDGD domain, which targets at reducing the gap between the existing MDGD approaches and the commercial game development practices.

In this chapter, I will briefly describe the background of my research as well as related work, pointing out the issues in the MDGD field, which motivates the research. I will also define a set of research goals and correspondingly create a roadmap to reach the goals. Finally, the structure of the whole thesis is presented, with a short description of each chapter.

1.1 Developing A Game Is Not A Game

“Developing a game must be as fun as playing a game”. This was what I thought about game development when I was a high school student in the early 90s. During my high school life, I used much of my leisure time in the computer lab, either playing or programming games. Creating my own games was very exciting, especially when they attracted more than half of the people in the small lab and they got immersed in my own developed games. The experience from game development satisfied me as much as playing through Final Fantasy 3 (FF3) or Romance of the Three Kingdoms 4 (RTK4), which blurred the border between developing a game and playing a game in my mind. Games have evolved so fast during and after my school years that in just a couple of years Role Playing Games (RPGs) like FF3 were replaced by Massively Multi-player Online Games (MMOGs), and turn-based strategy games like RTK4 were replaced by real-time strategy games. Game software was becoming more and more complex and could no longer be created by a single person or a small team of programmers. But my passion for game development remained, which eventually led me to the game industry in 2004. The company I joined was Ubisoft, a top game publisher in Europe, which had 9 studios over the world, and what I was working for was the third biggest one. The experience from developing real commercial games soon changed my mind about game
development: Yes, it is exciting, but it is way harder than gaming. My memory from being a game developer consists of intensive workdays, tight schedules, missed deadlines, unbeatable bug lists, and frequent inefficient communication among stakeholders. Game development today is very difficult, which is not only my personal impression, but also is suggested by studies such as [1], and the facts below may to some extent explain why:

*The requirements of computer games keep evolving throughout the development process.* Although “entertaining” or “fun” as the overall goal is unchangeable, the concrete functional requirements and the game design are always subject to change till the last day. This is due to the gap between the anticipated gaming experience and the actual gaming experience, which makes even an experienced game designer not be able to guarantee that an idea works before it is play-tested on the target machine. For example, I have worked in a team with 100+ developers for *Tom Clancy’s Splinter Cell: Double Agent* (Ubisoft 2006), and most of the people in the team had developed the previous titles of the game. The team could be considered quite experienced with the genre and the franchise. However, after about one year of development, the initial design was revealed not to be sufficiently enjoyable, so most of the game design and code were reverted, resulting in a project almost without any useful code and assets, and half time of the budget left. Consequently the development process for the rest of the project was full of pressure and doubt. Fortunately we still made the game a success for the cost of a significant delayed schedule.

*Games have to be developed through a creative process, which makes it difficult to directly transfer general software engineering practices.* In terms of its artistic nature, the development team has a majority of people from artistic instead of technical background. Artists work in creative processes, which is difficult fit with structured engineering work. In [2], Callele et al. argued that imposing too much structure on the creative process may be highly detrimental – constraining expression, reducing creativity, and impairing the intangibles that create an enjoyable experience for the customer.

*Communication is important for the success of game development, but it is difficult.* The success of developing a computer game requires effective communication in the multi-disciplinary development team. Prominent game designers may have a good understanding of underlying techniques and even can do some coding themselves, but this is not always the case. In practice, most gameplay ideas have to be explained by game designers to programmers, either through verbal communication or through written documentation. Documentation was highly recommended in [3, 4], since it both helps production and improving game design. But creating useful documents is difficult because the technical specifications of a game design story can easily reach 50 pages [5], and some designers insist that game design is something one cannot write down on a piece of paper [3]. So today’s game development still highly depends on verbal communication and weak documentation. Another problem is the stakeholders are speaking different languages, and this makes the communication even more difficult: a programmer may not understand the nature of a game design, and a game designer may not understand the technical solution produced by a programmer before a playable prototype is available.
There are challenges connected to the tools and workflow [1]. Commercial games today are complex software, which is almost impossible to be developed from scratch. Game engines emerged in the 90s, which provided high-level scripting languages, convenient game world creation tools, and industry-strength implementation of common technologies. Game engines have made the game development easier than before. However, as it was pointed out in [6], when the technology evolves, the demands for software also increase. According to today’s complexity of games helped by the engine tools and the scripting language, game software still requires a lot of development effort. An AAA title today (major game title) usually requires about 40 engineers working for 1-3 years to develop using a hybrid of scripts and native code. Moreover, due to the limitation of tools used, the game software usually grows huge in the end of the development, thus a clean build can take as much as one or more hours. Making binary and downloading on target machine also take time.

Due to the difficult nature, game development has a high possibility of failure. Moreover, the huge investment also has to be considered: The investment for an AAA title today can be as much as a Hollywood movie or more. This means that developing a game today is a business with high financial risk. Failure of developing a game can be a huge disaster to any company in the industry. This situation motivated my research, which aimed at developing new technologies and methodologies that can make the game development easier thus lowering the risk of game production. Model-Driven Development (MDD) was chosen as the direction of my studies. MDD has shown its effectiveness in the improvements of software productivity in various software domains, and game as a kind software is particularly appropriate for MDD. I will elaborate the reasons in the next section.

1.2 Why Model Games

Model-Driven Game Development (MDGD) is an emerging research field, which has attracted more and more researchers’ and practitioners’ interests in recent years. The philosophy of Model-Driven Development (MDD) is making models the core artifact of software development, and generating code fully or partially from models. MDD has some significant advantages in software development, which make it appropriate for some software domains. When it comes to game development, as Walt Scacchi pointed out in [7] domain specific languages that allow designers to be creative and expressive is important for future game development. Furthermore, I can point out following opportunities of applying MDD, which are particularly useful in solving the challenges discussed in Section 1.1:

1) MDD reduces the problem-implementation gap [6]. France and Rumpe argued that a significant factor behind the difficulty of developing complex software is the wide conceptual gap between the problem and the implementation domains of discourse, which is named as the problem-implementation gap [6]. Computer games are complex software and suffer from the same problem. To reduce the gap, abstractions need to be built and manipulated. The most common approach is to implement a library or framework and to use it through command-query API calls [8]. Game engines have been acting as such libraries or frameworks for about two
decades. However, developing a game engine is expensive, and the investment is supposed to be refunded by the cost deduction of developing games based on the engine. The more games that can be developed by the engine, the faster the engine generates profit. Hence, game engines have to trade abstraction level with generality, which leaves the problem-implementation gap open. For example, Unreal Engine has been used for constructing games in a number of genres other than the First Person Shooter it was targeting, including the popular third-person shooter Gears of War by Epic Games [9], while Torque 2D engine can be used for creating almost all genres of 2D games, from platform games to adventure games.

MDD has the opportunity here for reducing the remaining gap between game engines, and specific game domains, because MDD tools are relatively cheap to build using exiting MDD technologies such as meta-modeling languages and language workbenches. This means that building a specific MDGD tool chain for a specific game domain is economically more feasible than building a specific engine.

2) MDD makes prototyping of design ideas easier. The requirements of computer games come from the game design. Game design is creative work where many ideas are generated, tested and thrown away, while only a few ideas are implemented in the retailed product. The process of game design is necessarily iterative, or looping [4], and the iterative design emphasize play-testing and prototyping [10]. It is also advocated that prototyping of gameplay before adding some detailed artifacts is a good practice in [3]. But implementing the prototypes for validating designer’s ideas is a stressful, duplicated and disappointing process for programmers, because of: 1) tight schedule: the designer is always willing to see the running gameplay at once, 2) unstable requirements: the premature design changes very often, and 3) possibility of dropping: the whole work may eventually been thrown away without any contribution to the retailed game if the idea is considered not to be fun. The programmers have to continuously implement playable prototypes until the game design becomes stable, or the team has been in danger of failure to meet the critical milestones. The work for stabilizing game design is so time-consuming that a dedicated phase has to be introduced to the game development process, which is called preproduction [5]. Applying MDD can ease the preproduction phase, since it makes that possible that game designers’ ideas can be modeled and evaluated by the designers themselves, without going through a designer-programmer-designer route that overburdens the workflow, and can potentially waste development effort by misunderstandings.

3) MDD reduces the requirement to developers’ knowledge and experience. A factor that makes game development difficult is that computer games have some highly domain-specific requirements, which involves a lot of domain-specific knowledge [1], and this makes experienced programmers necessary. When MDD is applied, experienced and knowledgeable developers can work on the infrastructure development such as domain analysis, domain-specific language design, and tools, while junior developers and game designers can work on the game modeling. Another advantage of the finer granularity of work decomposition is that the developers can dedicate themselves to a specific development domain and thus be more proficient in it.
4) MDD can improve the communication efficiency. MDD’s opportunity here is that models, especially models created with domain-specific languages are formal specification of ideas. Due to the formality, communication using models can eliminate some mistakes in the communications with natural language. The models in domain-specific languages are also normally cheaper to create and easier to understand than textual documentation in natural language, since the domain-specific languages are supposed to be highly expressive and descriptive in the game domain. DSM can support the easier communication between software developers and domain experts [8] in general, and we believe it is also true in the game development domain.

The above-mentioned opportunities have motivated people to work in the MDGD field, and I will outline their contributions in the next section, pointing out the gaps, as the context of my research.

1.3 Research Questions

The concept of using models is not totally new in the game industry, since the content creation tools coming with game engines have a long history of using kinds of models. A typical example is the world editor or called level editor (e.g. Figure 2.4), which is the core tool in a game engine tool kit. A world editor provides a visual and even WYSIWYG environment for game world creation, i.e. modeling. However, the aspects that engine tools can model are still restricted to a narrow scope of the game software, while a large amount of elements such as complex behaviors and rules still have to be hand-coded, either with a native language or with a programming language. Moreover, the modeling tool itself is hardcoded, so it is difficult to be adapted to a new game domain.

Recently, researchers in the game community have proposed various Model-Driven Game Development (MDGD) approaches, for instance [11-15]. A thorough review of the related work has been done in the Ph.D. study to gather the state-of-art information. The results show that existing MDGD approaches are promising in improving the productivity and lowering the threshold of technology. On the other hand, as an emerging research field, MDGD still has many challenges, leaving gaps between the MDGD approaches in the literature and the game development in the industry. Such challenges include the target domain definition, tools interoperability, architectural support, lack of tools, and so on. More details about the review are presented in Chapter 2. Due to the time limitation, it is neither feasible to address all the issues in the thesis, nor is it necessary. The selection of the focus of the Ph.D. research is primarily based on my research interests and previous knowledge: Since I had experience as a game developer and a software architect, I am more familiar with and interested in the game engine and the software architecture fields. Therefore the following two challenges are selected as the major research focus:

1) Lack of the interoperability between MDGD tools and engine tools. The exiting MDGD approaches either ignored the game engine, or regarded the game engine
only as the run-time system. In both cases, engine tools are not considered as part of the development tool chain engine. Engine tools have significant advantages that make them implacable in near future: firstly they are industrial-strength implementation, which are more powerful and robust than in-house tools used; secondly they are built directly on run-time engines, thus can support some run-time features at design time; finally they have deep and tight relationship with the cultural and organizational situation of today’s game industry.

2) No Theoretical Base and Technical Support for Various Game Architectures. The software architecture of a computer game influences both the quality attributes and the functionality, and has therefore played an important role in game development. Today game architectures are getting larger, more complex and more sophisticated, evolving from stand-alone to massively multiplayer online game (MMOG) architectures. However, none of the existing work has addressed the game architecture in the modeling approach. How models work with different architectures remains a problem. Both the theoretical base and technical support for different game architectures are appealing to make MDGD accepted by the industry.

As a former game developer, I do understand the importance of game engines including their tool suites. If MDGD approaches cannot be integrated well with the game engines, the mainstream companies will undoubtedly reject them. Closing or at least reducing the gap can give MDGD better opportunities for being accepted by the game industry. Therefore one goal of my research is to reduce the gap by exploring the methodologies and technologies that can smoothly integrate the MDGD tools and game engine tools.

The diversity of game architectures is another important challenge that prevents MDGD from being used in mainstream game development. The game market was once full of single player games and multiplayer games on shared screen, which used the simplest single-software architecture. But after 40 years of evolution, the game architectures have been diversified. Although the single-software architecture still has some market share, client-server and multi-server architectures are becoming more and more popular. Peer-to-peer architectures also attract researchers’ attention since it can compensate some significant drawbacks of client-server architectures. Furthermore, various hybrid architectures are proposed in order to take advantages from both client-server and peer-to-peer architectures. However the existing MDGD approaches only considered standalone software architecture, while the architectural diversity of modern computer games is overlooked. How can the architectural characteristics be reflected in models? What changes have to be made in models in results from the changes in the game architecture? How can code be generated for different architectures? As long as these questions remain open, practical use of MDGD remains is still questionable. Thus, another goal of my research is to solve these questions, in order to reducing the gap between the existing MDGD approaches and the various game architectures.

To summarize, the goal of my research is a synthesis of the above two goals, which can be described as below:
The goal of my Ph.D. research is to push the Model-Driven Game Development methodology further towards the practical use in commercial game development, by proposing methodological and technical solutions to reduce two gaps: 1) the gap between the MDGD and engine-based development, and 2) the gap between the existing MDGD approaches and the diversity of game architectures.

The research goal can be further decomposed into a set of research questions, which are presented in the rest of the section:

**RQ1: How Shall MDGD be integrated with game engine-based development?**

This research question addresses the methodologies and technologies that integrate the model-driven game development in academic research and engine-based game development in industrial practices. Some of the existing MDGD approaches are aware of the game engine. However, their methods of integrating game engine with MDGD have significant drawbacks: They regarded the game engine only as the run-time library, whilst the engine tools as an important part of the engine were overlooked.

To answer RQ1, I will firstly review Model-Driven Development methodology in general, and investigate the existing MDGD approaches in particular, which provides background information for the next step of the research. Then I will propose a MDGD approach that takes full advantages of engine-based development to MDGD by integrating both the run-time engine and the engine tools.

RQ1 can be further decomposed into two research questions on two aspects:

**RQ1.1: What is a general model-engine integrative approach that is not restricted to a particular engine or model-driven tool?**

The research question is about the methodology aspect. The answer to the question should be a general MDGD approach that is independent of the particular game engines and MDD tools. The approach aims at complementing the game engine with the MDD technologies instead of replacing the engine. With the approach the modification to the traditional engine-based development methodology will be minimized, and the cost of the transition of the development process will be lowered, so it will be easier for MDGD to be accepted by the industry.

**RQ1.2: How should we implement an integrative approach with a specific engine and MDD tool?**

The question is on the implementation aspect. The answer to the question should be implementing a concrete modeling language and a tool chain that are successfully integrated with a specific game engine. The language and tools should validate the integrative approach on one hand, and provide a useful technical solution on the other. To evaluate the answer, a prototype will be developed using the modeling language and the tool chain.
RQ2: How Shall The Architectural Diversity Of Games Be Addressed in MDGD?

The research question is about how to improve the existing MDGD approaches with adding the support for diversified game architectures, on both methodological and technological levels. The question needs to be decomposed into a set of questions presented in the rest of the section, and several artifacts will be created to address them respectively.

RQ2.1: What is the current situation of the architectural diversity of games?

The question is about what software architectures are used in game software, and their characteristics as well as popularity. Building relevant knowledge of game architecture domain is the main purpose of the question. To answer the question, it is necessary to investigate the game architecture domain thoroughly, in order to know what architectures are important to be supported by MDGD, as well as their characteristics. The answer to the question also provides the necessary information, in terms of which conceptual models can be created for the next question.

RQ2.2: What is an appropriate conceptual framework of game architecture that is useful in MDGD?

This question is about the creation of a conceptual framework that helps describe the characteristics of game architectures in MDGD. The framework should lie on a proper abstraction level that its concepts are closer to the computer game domain than the concepts from the general software architecture domain. Therefore the concepts included by the framework should be a good theoretical base for defining both the syntax and the semantics of modeling languages.

For the sake of its abstract nature, the conceptual framework will not and should not only address MDGD, and ideally it is a theoretical tool for describing and analyzing game architectures in general. Other applications of the conceptual framework should also be discussed in answering the question.

RQ2.3: How can the architectural concerns be addressed with a MDGD approach based on the conceptual framework?

This question is about the specific application of the conceptual framework of game architectures in MDGD context. To answer the question, a MDGD approach should be developed, which ideally is an extension of the integrative approach discussed in RQ1. Apart from the integration with the game engine, the approach should support game architectures other than the standalone software architecture that the existing MDGD approaches solely support.

The architectural aspects should be specified using the conceptual framework discussed in RQ2.2, and this will validate the usefulness of the conceptual framework.

RQ2.4: How can we implement a specific modeling language and tool chain addressing game architectural concerns?

This question is about the implementation of MDGD approach discussed in RQ2.3. The
answer to the question should be an extension to the modeling language and tool chain discussed in RQ1.2. The resulted MDGD tools include modeling language, code generator, domain framework and so on which are not only integrated with a game engine, but also support the architectural aspects being specified at the model level.

The language and tool chain validate the quality of the MDGD approach proposed in answering RQ2.3, and they also justify an individual contribution as useful tools for game development.

1.4 Research Methodology and Design

The Ph. D. research is carried out following an iterative approach, where two iterations are performed. For each iteration, the research follows a top-down method, where the theoretical and methodological questions will be addressed first, then the technical questions, and finally the prototypes are developed. The major research methods used in the research include Literature Review, Systematic Review, and Design Science.

1.4.1 Research Methodology

Design Science is one of the main methods used in the Ph. D. research. The design-science paradigm has its roots in engineering and the sciences of the artificial [16]. It is an important research method in the information systems field. Design Science is inherently a problem solving process[17]. It involves the creation of new knowledge through design of novel or innovative artifacts and analysis of the use and/or performance of such artifacts along with reflection and abstraction—to improve and understand the behavior of aspects of Information Systems [18]. The artifacts created in the research include the conceptual framework, the MDGD approaches, as well as the modeling language and tools. The artifacts are evaluated by analyzing the use with prototypes, which will be detailed in the next section.

A review of prior, relevant literature is an essential feature of any academic project. An effective review creates a firm foundation for advancing knowledge. It facilitates theory development, closes areas where a plethora of research exists, and uncovers areas where research is needed [19]. Literature Review is used in the Ph. D. study for investigating the MDGD domain, in order to summarize the existing MDGD approaches and identify the gaps as the candidate research questions.

Systematic Review is a structured literature review aiming to provide an exhaustive summary of current literature relevant to a research question [20]. Systematic Review features the transparent literature search process, explicit inclusion and exclusion criteria for sources, use of statistical techniques, and etc. Systematic Review is used in the Ph. D. research for evaluating the conceptual framework of game architecture. The structured review produces a complete view of game architectures in the literature, and by fitting the architectures into the conceptual framework, the descriptiveness of the framework as a theoretical model is validated. I choose the Systematic Review instead of the ordinary Literature Review because it provides better literature coverage, which provides more reliable results.
1.4.2 Research Design

Figure 1.1 shows the research process with two iterations, where the first iteration focuses on RQ1 and the second iteration focuses on RQ2.

![Figure 1.1 Research Design]

The dotted line in Figure 1.1 shows the artifact-evaluation relationship, where the arrow points to the artifact to be evaluated. The circles connected to the activities represent the expected contributions, which are described in Table 1.1.

<table>
<thead>
<tr>
<th>Table 1.1 Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
</tr>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

The first iteration starts from a survey of the MDGD research field to build the necessary knowledge and to gather the context information for further research. The
main research method used here is literature review, and the contribution made is a state-of-art report. Through the review, a couple of gaps are identified in the existing work, and the engine-integration is selected as the main goal of the first iteration.

Then I will propose a method integrating MDGD with engine-based development. The work will be inspired by related work reviewed in the first activity, such as [21]. The main research method used here is design science.

To evaluate the integrative method, I propose to create a modeling language: RAIL and its tool chain in the technology development activity of the first iteration. The main research method used here is still design science. The experience from creating the language and its tool chain will provide feedback to the methodological level development, which help improve the methodology. The integrative method will be evaluated by the implementation and the use of RAIL. On the other hand, RAIL is also an individual contribution as a MDGD tool for modeling gameplay of action/adventure games.

Finally a prototype game: Orc’s Gold is developed to evaluate the usefulness and appropriateness of RAIL and the language tools. Hopefully the evaluation will prove the effectiveness of RAIL in gameplay modeling.

The second iteration starts from the conceptual level work, where I will propose an architectural framework, which can be used in game architecture classification, analysis and design. The framework serves the theoretical base of other research in the second iteration. The main research method here is design science.

Then the architectural framework will be evaluated through classifying and analyzing 40 game architectures that are described in the literature. Systematic review is used here as the main research method. The systematic review provides good literature coverage and it qualitatively evaluates the usefulness of the framework as a tool for classifying and analyzing game architectures. Moreover, the review also provides important information about the characteristics and popularities of the architectures for the MDGD methodology and technology development.

When the architectural framework is ready, supporting architectural features in MDGD is relatively easy at the methodology level. By integrating the concepts from the conceptual framework to MDGD, the architectural aspects can be specified in the game models. The result of the integration is an architectural method for addressing architectural concerns in MDGD. The main research method here is design science. The quality of the conceptual framework as a theoretical base for MDGD is evaluated by the use of the architectural method.

Furthermore, RAIL and its tool chain will be extended to RAIL2 using the architectural method to support various game architectures, and design science is still used as the main research method. The usefulness of the architectural method is evaluated by the implementation and use of RAIL2. Apart from the method evaluation, RAIL2 is an
individual contribution as a tool for gameplay modeling that supports various game architectures.

The whole research will be finalized by developing and evaluating a prototype game: Orc’s Gold 2, which is implemented with both MDGD tools and game engine tools and supports three game architectures. The evaluation of RAIL2 is done through analyzing the prototype.

### 1.6 Thesis Structure

The rest of the thesis is structured as follows:

In Chapter 2, I present the state of art of model-driven game development field, and identify gaps that motivate my research.

To reduce the gaps, I define a set of goals for my research, which are further elaborated with a couple of research questions. To find the answer to the questions, I used an iterative process, which is described in Chapter 3. The research process included two iterations, and I followed a top-down approach in each iteration, which starts from theoretical or methodological level development then down to tools design and prototype development.

The first iteration of my research focuses on integrating game modeling into the existing game development process. In Chapter 4 I present Engine Cooperative Game Modeling (ECGM), a new game modeling approach to start the first iteration, and the approach features the cooperation between engine tools and modeling tools. The approach minimizes the modification to the existing game development process, while still takes the advantages from MDD, so it is more practical than the exiting MDGD approaches.

To validate the ECGM approach, I created a modeling language for high level AI modeling in 2D adventures, which is named Reactive AI Language (RAIL). The language is described in detail in Chapter 5, together with a game prototype named Orcs Gold. The high level AI of the game is developed by RAIL tools, and I present the game as a case study to finish the first iteration of research.

In the second iteration, I focus on handling architectural diversity of games in MDGD. The second iteration of my research starts from Chapter 6, where I propose Game Worlds Graph (GWG) framework. The framework can be used to understanding existing game architectures and design future architectures. It serves the theoretical base of my MDGD approach addressing architectural diversity of games.

In Chapter 7, I review the existing game architectures with the GWG framework. The complex and diversified nature of game architectures make the architectural concerns critical in game modeling, which provide the context of further research.

The results of the second iteration are presented in Chapter 8, where we extend RAIL to RAIL2. The language now can handle the architectural concerns, supporting various
game architectures when model the games. The language is an extension of RAIL, which remains the feature of cooperation with game engine. GWG framework serves corner stone of the language, by providing the conceptual abstraction of game architectures. Orcs Gold2, a game developed with RAIL2 is also discussed in the chapter as a case study.

By summarizing the two iterations of the research, I conclude the thesis in Chapter 9, where the results are presented and contributions are highlighted.
2 Towards Model-Driven Game Development (MDGD) Industrialization

Model-driven game development introduces model-driven methodology to the game domain, shifting the focus of game development from coding to modeling in order to make the game production faster and easier. The research on MDGD is concerned with both the general model-driven software development methodology and the particular characteristics of game development. MDGD implies changes in many aspects of game development, from the technology to the workflow, and from the organization to the development team culture. This chapter presents the state of art of MDGD research, as well as an introduction to the MDD methodology in general. The chapter is based on a survey of MDGD approaches in the literature with a main focus on the challenges related to the industrialization of MDGD. Gaps between the academic studies and commercial game development were revealed in the review, which are summarized in this chapter.

2.1 Model-Driven Development: Basic Concepts

The central concept in the MDD domain is the model. In [22], model is defined as “a simplification of a system built with an intended goal in mind”. A similar definition can also be found in [6]. This means that theoretically many things are in the scope of a model. E.g. Favre argues in [23] that an article is itself a model: a model of the topic that it is about. However, the pragmatic scope of model is usually narrower. For example: Kelly and Tolvanen describe the developers understanding of models and code as “models are used for designing systems, understanding them better, specifying required functionality, and creating documentation. Code is then written to implement the designs.” [24] They distinguish models and code, which is necessary for consolidating the theoretical base of MDD. Otherwise will MDD cover the traditional development methodology in terms of the provision that code is a kind of a model. Furthermore, in MDD, models are more than just documentation of the system: They can be transformed into a system, or they are the system itself.
Model-Driven Development and Model-Driven Software Development (MDSD) are often used interchangeably in the software engineering community, although the former can mean more than just software development. In this thesis, MDD means the same as MDSD. Various definitions of MDD have been proposed in the literature [25-28], where MDD can be defined as a software development methodology with following characteristics:

3) **Models are the focus**: MDD focuses on the models rather than the code, and the models are the major artifacts of the software development.

4) **Models are formal**: Models in MDD are formal thus can be transformed into the software automatically, or the models are executable.

5) **Models are on a high abstraction level**: Models are created with a modeling language at a higher abstraction level than the programming language, thus reducing the problem-solution gap.

An term similar to MDD is Model-Driven Engineering (MDE), which sometimes refers to the same concept as MDD in research papers, e.g. [29, 30]. But in [6], France and Rumpe enrich the meaning of MDE by distinguishing the development models and the runtime models. The two models further lead to two directions of research with different main research questions. The research focusing on development models is mainly concerned with how can modeling techniques be used to tame the complexity of bridging the gap between the problem domain and the software implementation domain, while the research focusing on runtime models is mainly concerned with how can models be cost-effectively used to manage executing software, and how can models be used to effect changes to running systems in a controlled manner? We agree with their opinion, which implies a broader scope than MDD, covering development of software by models as well as run-time management and control of software with models.

Models in MDD can be created with either General Purpose Languages (GPLs), or Domain-Specific Languages (DSLs). UML[31] is the most popular modeling GPL standardized by Object Management Group, which was developed to be able to model all kinds of application domains [24]. However people in the MDD community have pointed out some drawbacks of the language, e.g.[32]. Especially the one-size-fits-all design has been criticized for lowering the abstraction level [24]. The 2.0 version of UML supports semantic variation points and profiles as two forms of extensions, which improved its domain appropriateness [33]. But the support for domain-specific abstractions is still weaker in UML compared to another approach of MDD so called Domain-Specific Modeling (DSM). DSM uses the modeling language developed for a specific application domain to solve the problems within the domain. DSM was claimed to have better domain appropriateness, restricted semantic scope, better support for generating code, and increased domain-specific reuse of components [34], and was also reported better than GPLs, regarding to the improvements on software productivities [24]. DSM also has its drawbacks, where an important one is the possibly huge investment on developing a DSL. Although the time to implement the DSL can be short, the expected time to benefit from it can decrease the investment interest [24]. An
extensive consideration must be made on whether a new DSL for a problem should be created or not. In [35], the problem is discussed through identifying a set of decision patterns from the cases of applying DSM. As a summary, Figure 2.1 shows the concepts related to MDD, and their relationships.

![Figure 2.1 MDD Concepts and Their Relationships](image)

I have introduced MDD in this section, and the remaining will focus on applying MDD specifically on the game development domain. France and Rumpe’s state that research on both extensible GPL and DSL will provide valuable insights, and both of the approaches can play a key role in MDE [6]. According to this, both GPL and DSL approaches will be covered in this chapter. The term Model-Driven Game Development (MDGD) will in this thesis denote MDD in the game development context, which is also used in other articles in the game modeling community, e.g. [11, 36].

### 2.2 An Overview of Exiting MDGD Approaches

The game industry has actually a long history of using models. A typical example is the level model created with the level editor, which provides a visual environment for game world modeling. However, the game elements that engine tools can model are restricted to a narrow scope of game software, which are mainly assets like level data and presentation data. A lot more elements such as AI, control, rules still have to be hand-coded, either with a native language or with a scripting language. Some state-of-art game engines do provided visual modeling tools for creating script code, for example, Unreal Kismet [37], but these tools are not taking full advantages of MDD, such as use of meta-models and language workbenches, which makes it difficult to be adapted to a new game domain.
To address the drawbacks of existing engine tools, researchers in game community have proposed various MDGD approaches, and Table 2.1 shows the representative work in literature labeled with an ID that makes it simple to refer to them later in this chapter. There are more articles about game modeling in literature for example [38-41], but they are not included in this chapter because we only look into the work within the MDGD scope that was documented with sufficient details. This means that as the central artifacts of game development the models must be described in a formal modeling language whose syntax and semantics were well-defined. The language is not only intended for a game design specification but also executable (or executable after transformation), and it is supported by a tool chain and/or created by a language workbench.
Table 2.1 MDGD Approaches in Literature

<table>
<thead>
<tr>
<th>ID</th>
<th>MDGD Approach</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC1</td>
<td>Modeled 2D Platformers with UML, and the models covered <em>Structure</em>, <em>Behavior</em> and <em>Control</em> of games</td>
<td>[11, 36]</td>
</tr>
<tr>
<td>A1</td>
<td>Proposed a model-driven approach based on UML profiles for modeling board games such as chess</td>
<td>[42]</td>
</tr>
<tr>
<td>HO1</td>
<td>Proposed a graphical DSL: Eberos GML2D for modeling <em>all</em> 2D games</td>
<td>[43]</td>
</tr>
<tr>
<td>M1</td>
<td>Discussed a case study about using DSL for developing RPG games for mobile phones</td>
<td>[44]</td>
</tr>
<tr>
<td>PH1</td>
<td>Proposed a MDD approach featured a DSL named MML for interactive multimedia applications (typically games) development. The approach integrated MDD tools with Adobe Flash.</td>
<td>[21] [45, 46]</td>
</tr>
<tr>
<td>F1</td>
<td>Proposed a graphical DSL named SharpLudus for modeling 2D adventures</td>
<td>[47-49]</td>
</tr>
<tr>
<td>S1</td>
<td>Proposed a textual DSL for modeling general games, which is based on a feature model and XML.</td>
<td>[50]</td>
</tr>
<tr>
<td>FR1</td>
<td>Proposed a textual DSL named PULP for modeling interaction HTML5 applications including games.</td>
<td>[51]</td>
</tr>
<tr>
<td>MS1</td>
<td>Proposed <code>&lt;e-Game&gt;</code>, a model-driven approach featuring a textual mark-up language for graphical adventure games.</td>
<td>[52, 53]</td>
</tr>
<tr>
<td>MV1</td>
<td>Proposed a MDGD approach following the idea of software factories, and suggested language workbenches as the central tool of the approach.</td>
<td>[54]</td>
</tr>
<tr>
<td>F2</td>
<td>Proposed an iterative process for creating game software product lines, where domain specific modeling played the core role.</td>
<td>[55, 56]</td>
</tr>
<tr>
<td>WM1</td>
<td>Proposed a workflow to integrate DSM into the game development process.</td>
<td>[57]</td>
</tr>
</tbody>
</table>

Table 2.1 mainly includes two kinds of work, where one kind represents most of the studies focusing on the technological aspects of MDGD typically proposing a modeling language and maybe the implementation of the language, and another kind focusing on the methodological aspects of MDGD typically discussing the process and management level issues. The former includes [A1], [M1], [S1], [MS1], [RC1], [HO1], [PH1], [F1], and [FR1]. The latter includes [F2] and [WM1]. [MV1] is a study in-between.

[RC1] describes how the authors model 2D platformers using UML. The model included three diagrams, which are structure diagram, behaviour diagram and control diagram. Both static and dynamic aspects of game software were specified in the model, and an automatic generator is implemented with MOFscript transformation language that can generate C++ code. [A1] is also a UML-based approach, which addresses the board game domain. The authors identified 11 core concepts of board game by
analyzing the domain and established a meta-model for it using UML. The meta-model was used to create a UML profile, and the two games Chess and Backgammon were modeled using the UML profile. The models were used to generate code with MOFscript tools. They also tried to use Ecore as a meta-modeling language, and defined an XML schema (XSD) for the meta-model of board games. The Chess game was re-implemented with XML conforming to the XSD, and was transformed to text description of the game with the oAW framework.

[FR1] describes PULP, a textual DSL for modeling interactive HTML5 applications, typically games. Basically the DSL supports two aspects of game modeling: 1) the game world modeling with screen primitives such as screens, images, text 2) the objects interaction modeling with the action chain concept. PULP was implemented using the Eclipse framework. Specifically, Xtext was used to create the text editor, and Xtend2 was used to implement the model to HTML 5 code generator.

[S1] describes a MDD approach for generic game development. The approach starts with GDS, a game feature model, which tries to describe the commonalities and variabilities among different games. Then GSL, a textual domain specific language is defined, which is based on XML by means of specific tags. The language is able to model game world in terms of specifying the attributes of game objects comprising the world, and also able to model the framework of behavioral aspects of games. The model can be translated into classes integrating with a “meta-programming framework” and the game framework (for example a game engine). To make the game fully functioning, the generated classes need to be implemented to communicate with the game framework.

[F1] describes SharpLudus, a software factory for developing the product line of 2D adventure games, which features two DSLs: SLGML for game world modeling and Head-Up Display (HUD) creation DSL for modeling the presentation of game information. SLGML is a graphical language supporting a set of game concepts such as entity, event, sprite, etc. Game flow can be visually specified in a way similar to state machine diagram, where the states are represented with in-game pictures. C# code can be generated from the models, and it runs on a simple game engine specifically created for the software factory. Developers can also write custom code in addition to the generated code, which can be integrated together into the complete game software.

Hernandez and Ortega believe that 2D game is narrow enough as a target domain, so they proposed Eberos GML2D [HO1] as the language for modeling all 2D games. The language supports Sprite, Animations and Entities as the basic constructs for modeling the static elements, while state machine with one concurrent level of execution is used together with embedded target code for modeling the dynamic aspects of games. The language also provides services like collision detection, resource management, and messaging.

[M1] describes a DSM approach for developing Role Playing Games (RPGs) for mobile phones. The authors analyzed both the problem domain (the RPG game concepts) and the solution domain (the target game engine) to gather information for the language and
tools design. The language and tools are built on EMF, GMF, and ALPiNA [58], and the graphical DSL is capable of game world modeling. Automatic code generation is implemented through a two-step process including a model-model transformation, and a template based model-code generation. ALPiNA-based semantic validation is also implemented through a similar two-step process. Intermediate languages are created in the code generation and model-checking implementation to support the indirect processes.

Multimedia Modeling Language (MML) is a graphical language proposed by Andreas Pleuss and Heinrich Hussmann for modeling multimedia interactive applications [PH1], while the video game is a typical example. The MML provides language level support for modeling three aspects of software: the structure by the UML class diagram with the extension for multimedia objects, the multimedia user interface by specific notations, and the interaction by the UML activity with the extension in integration user interface objects. The tools for the language is implemented with Eclipse Modeling Framework (EMF) technologies, and further integrated with a multimedia authoring tool, Adobe Flash, through model transformation with Atlas Transformation Language (ATL). So Flash- based multimedia interactive applications including games can be developed through the combination of using the language tools and using the Adobe Flash.

The <e-Game> project [MS1] established a MDGD approach for graphical adventure game development. The approach featured the use of textual mark-up language that should be developed in an iterative manner throughout development of similar games. The mark-up language was used to annotate game storyboard written by Game Writers, and the resulted document described both the structural and behavioral aspects of the game in question. The annotated document can feed a <e-Game> engine, which loads the document into the engine as a tree, and associates a formal operational semantics to the tree. When the associating semantics are in turn evaluated by the engine, the game runs and becomes interactive.

[MV1] proposed a game development approach following the idea of software factories, which combined the software product line, component-based development and model-driven development. Maier and Volk discussed the roles and workflow in the approach, and distinguished the product line developer and the product developer. The former focuses on the software factory development, which is mainly about creating domain specific languages and the corresponding tools. The latter focuses on the actual game development using the tools produced by the product line developer. The authors emphasized the use of a language workbench in the approach, which can efficiently generate the tools for domain specific modeling and allows the flexible re- specification of the tools for game development. Language workbenches make it possible to create DSL tools for very narrow domains thus increase the productivity impact of MDGD.

[F2] suggested an iterative process for developing game software product lines (SPLs), which featured domain specific modeling and automatic code generation. The process combined top-down and bottom-up approaches, where the problem domain is elaborated in alternations with the solution domain. The process is iterative, and in each iteration the following four phases are defined: Game Domain Envisioning, Game
Domain Analysis, Application Assets Creation, and Development Assets Creation. They further validated the process by revisiting [F1], and creating AradEx, a SPL for 2D arcade games.

A MDGD workflow was proposed in [WM1] for integrating DSM into the game development process. The workflow followed a bottom-up approach starting from language requirements to reference artifacts, then up to language definition and finally the domain-specific programs. The authors also defined roles in the workflow, and discussed the responsibilities of each role. To show the usefulness of the approach, the author described a 2D Point & Click adventure game created through domain specific modeling, and introduced the DSL for modeling the game.

In the rest of this chapter I will discuss the findings and inspirations in reviewing the MDGD approaches, and then make a short summary.

2.3 Domain Definitions

An essential value of MDD is the raised abstraction level, which allows specifying solution with problem domain concepts thus making the solution simpler and easier. To this end, both the modeling language and the generator need to be domain-specific [24]. This is because the broader domain implies the more high-level concepts, which result in the more complex language and corresponding generator. To be successful in practice, MDD approaches must find the balance among the abstraction level, the complexity of the language, and the broadness of the domain. This means that choosing a proper domain is a necessary step to make MDGD useful. An improper domain selection tends to fall into two directions: too narrow or too broad domain.

Too narrow domain may result in the success of MDD in one project, while the failure from an overall economic perspective. This is because it is very difficult or even impossible to reuse the language and tools developed for one project in another due to the domain restriction. Thus the cost (usually expensive) for building the MDD tools must be returned from a single project, which is not always possible. On the contrary, too broad domain leads to three possible issues with the DSL as listed below:

1) Unmanageable number of language constructs. The broad domain brings in a large number of concepts that the DSL has to support. When the number reaches a certain point, the complexity of the language will make the development and maintenance impractical.

2) Low abstraction level. To control the language complexity in a broad domain, the development organization may choose to sacrifice the abstraction level. Although the number of language constructs decreases, the DSL may become less useful. In extreme cases, the abstraction level can be as low as the third generation language, eliminating the advantages of MDD.

3) Poor descriptiveness of the DSL. In a too broad domain, if the language wants to keep both high abstraction level and manageable number of constructs, it has to sacrifice the descriptiveness, i.e., making an incomplete language. The DSL can work for one project but may not work for another one in the same domain, due to the limited expressiveness.
The MDGD approaches in literature chose different domains in different way. All of the approaches selected a horizontal domain that is characterized by the game genre or the gameplay features. The broadest horizontal domain may appear in [S1], where the approach was claimed to support “generic game development”, meaning that all kinds of games can benefit from the approach in their development. However, games from different genres are so diversified, e.g. a first person shooter has a totally different domain concept compared to a Solitaire card game. To provide language constructs for all the high-level concepts of all games is not realistic. The solution proposed in [S1] is to provide general constructs comprising a modeling framework, and when model a specific game, the constructs have to be elaborated, new concepts are defined as XML tags (the DSL is XML based) and organized with the framework. A generic code generation script will generate class skeletons and some implementation of the classes, and the game specific code still have to be developed by hand. The DSL in [S1] actually is more like a language for modeling the framework of a game, instead of modeling concrete and complete game software. [FR1] is another approach with a very broad domain, which was claimed to support “interactive HTML5 applications”. Looking into the language, it actually put its focus on game world modeling, and interaction modeling. The concepts provided do cover important aspects of common HTML5 games. However, if we look into specific game genres, many core concepts are missing, e.g. physics concepts for platform games, AI for strategic games, etc. So either the DSL needs to be extended or the manual coding is needed when developing a specific game. [PH1] aimed at a similar domain as [FR1] that is games based on Adobe Flash. For the same reason as [FR1], the actual domain concepts that [PH1] directly supported were limited. Hernandez and Ortega argued that 2D game is a domain narrow enough for MDGD, and they created Eberos GML2D [HO1] to support all 2D games development. The language included generic concepts for 2D games such as sprites, entities, animations, state machine, and messages. They also validated the approach by modeling and generating two games. However, the two games are both simple platform games. The question still remains whether the approach scales with the complexity and diversity of commercial 2D games or not. 2D games actually have many genres from board games to MMORPGs. Different genres have distinct core concepts, which were rarely supported by the language. For example, we cannot find the constructs for modeling strategic AI (strategy games), conversation (adventure games), and rules (card games). Although the DSL captured the commonalities of the 2D game domain, the details of genre features in the domain were overlooked. The details are however what matter in DSM practices [24]. We have to keep in mind that “developers often create a language that’s too generic for its domain, with concepts and semantics that are either too few, too generic, or both” is one kind of worst DSM practices[59].

Apart from the above broad domain definitions, most of other MDGD approaches chose a specific game genre as the target domain. [RC1] was intended for 2D platform game development, which is a kind of action game featuring the player character travels around the game world by jumping among platforms. [A1] targeted board game development, which are computerized traditional multiplayer games played on tabletop, e.g. chess. [MS1] and [WM1] both chose the graphical adventure domain, which are games featuring pointing and clicking graphical objects in a statically presented
background, talking to characters, collecting clues, and etc. Among these genre-based domain definitions, [M1] provided the narrowest genre scope that is RPG for mobile phones. This definition both restricted the genre and the running platform. The genre-based domain definitions seem more realistic than the holistic domain definitions like “general games”, but they still suffer at least one of the following problems:

1) The domain supported is narrower than the domain claimed to support. The authors of the approaches tended to underestimate the scope of the genre. E.g. [A1] targeted board games, however, it actually supported the board games in Abstract Strategy category according to the taxonomy of boardgamegeek.com, which includes 84 such categories. Looking into the approach, we can see that popular board games like Agricola (Economic category), Magic The Gathering (Card category), as well as many other board games can hardly be supported by [A1]. Possibly it is due to the definitions of game genres usually vague and subjective. When people pick a genre name to define the domain for convenient considerations, they actually altered/adapted the genre definition to the context of the approach.

2) The DSL was too general by including common concepts for the game genre, but overlooked the specific details distinguishing game families in the genre. This is the same problem as some approaches targeting general games. But it is not as significant as general approaches, since focusing on one genre did narrow the language scope. For example, [RC1] was aiming at 2D platform games, and the concepts supported by the DSL such as entities, player characters are convincingly important for the genre. However if you start to model a complete commercial platform games with sufficient features, you may find yourself building a house with just one axe – yes you can, but too difficult: no inventory concepts, no physics concepts, no camera control concepts, etc.

3) Despite of the appropriateness of the horizontal domain definition, the vertical domain was not well defined. None of these approaches generated all elements of the game software, some artifacts still have to be developed using traditional methods such as coding and digital content creation tool. But there was no explicit specification about what aspects were conveyed by MDGD and what aspects were left to other methods. We call the specification Vertical Domain definition, in opposite to the Horizontal Domain definition. Without the vertical domain definition, it is difficult to know what the MDGD approach can actually do to a specific game project.

Different from the genre based definitions above, [F1] presented a more systematic definition of the target domain, where the game domain were outlined as Adventure games, and a detailed description from nine perspectives corresponding to nine key features of such games was presented, in addition to the genre keyword. ArcadEX is another approach proposed by Furtado as the case study of [F2], which used a feature model in addition to the keyword: 2D Arcade Game to characterize the target domain in a more systematic way. The model includes 150 features describing the domain’s commonality and variability. These two definitions are much more precise and useful
for evaluating the approaches in a concrete project context, although they still lack the vertical domain specification.

Lastly, the case study in [MV1] went for another extreme that followed “one game at a time” principle thus further narrowed the domain down to a single game, where a vertical domain specification was still not provided. Creating DSL and tools for one game may be economically unfeasible, so they promoted the language workbench as the core for MDGD environment. As a summary, Figure 2.2 illustrates the domain broadness of the reviewed approaches.

![Diagram of domain scopes]

**Figure 2.2 Domain Scopes of The Reviewed MDGD Approaches**

The approaches in Figure 2.2 are divided into four tiers, the higher the tier is, the broader the target domains are.

### 2.4 What Is A Proper Domain Definition?

First of all, a proper domain definition of a MDGD approach must include both the *horizontal domain* and the *vertical domain*. The horizontal domain defines the characteristics of the games that the MDGD approach is intended to support. The vertical domain defines what aspects of the game software will be modeled and what are left to traditional development methods. The horizontal domain provides the view from the feature perspective, and the vertical domain provides the view from the software architecture perspective. The existing game genres such as RPG, Platform games, RTS, etc. serve the initial specifications of horizontal domains, but they need to be elaborated to justify a useful MDGD domain definition. Furtado suggested a set of core dimensions for defining game domains in [55], including player, graphics, flow, entities, events, input, audio, physics, AI, networking and any custom dimensions. These dimensions comprise a base for creating a systematic framework for the formal domain specification.

The vertical domain definition is still an untamed problem in the MDGD community, which is about how to classify the aspects of game software in the MDGD context and which of them are going to be built with MDD. Reyno and Cubel proposed an initial specification in [38], which included the social context, internal game structure, and rule definition. They also stated that other aspects are left for future research. If we look into the existing MDGD approaches, we can append more aspects including game world
composition, game object behavior, control system, etc. There are still some un-
mentioned but important aspects such as artistic assets, presentation, on-screen GUI,
narrative flow, etc. Not all of these aspects are necessary for one MDGD approach,
because traditional methods such as programming are suitable for some of them. For
example, low-level AI (e.g. path finding) can be implemented effectively with code.
The decision on vertical domains can be different in different project contexts, and the
developers who are familiar with the solution domains are recommended to participate
in the decision process. This is because they are the persons who knows the game
engine or game framework resources [50]. The vertical domain definition is important
because it firstly limits the DSL to a proper scope, and secondly helps the stakeholders
anticipate the benefit of MDGD by knowing what it can do, thus improving the
confidence of adopting MDGD.

Another suggestion is to further narrow down the horizontal domain. From my
knowledge and experience in the game industry, my opinion inclines to the "one game
at a time principle" [54], but I would advocate the DSL reuse with modifications to
lower the average cost of DSL creation. This is because: 1) The computer game industry
is innovation-driven and unique characteristics justify the existence of successful
games. In practice a game and its sequels can be quite different in terms of features and
complexity, for example Super Mario Bros I and Super Mario Bros III (Nintendo), and
WarCraft II and WarCraft III (Blizzard Entertainment). 2) Successful games tend to
absorb nutrition from various genres, instead of being constrained to one genre.
Looking into recent games like The Legend of Zelda: Twilight Princess (Nintendo
2006), such games contain a wide variation in gameplay including elements from puzzle
games, fighting games, adventure games, and board games Even the old games like
Contra (Konami 1988), cannot be easily classified into a single genre, since although
four out of six levels are fit into the platform genre, the rest two are 2.5D shooter.
Imagining creating Contra today with a DSL aiming for the 2D platform game, the two
innovative levels may have to be excluded. The intention of MDGD is to release game
developers from error-prone tasks thus put more resource in innovative work. But if we
make a DSL aiming for a single genre or even a broader scope of games, we are in
danger of using MDGD against its intention: restricting the creativities of game
designers instead of freeing them.

Creating a DSL for a single game can maximize the game design possibilities, but the
cost becomes a problem. [54] advocated the use of a language workbench, which is a
viable solution. I want to point out another solution that is reuse with modifications:
DSLs created for one project can be stored in a language repository with a specification
about the horizontal and vertical domain definitions. In the pre-production [2] phase of a
new project, the DSLs in the repository are evaluated and a reference DSL will be
selected as the base for modification according to the new game design. In the pre-
production phase, prototypes are constructed by coding, and the code is the solution
domain input to the DSL design following a two- direction methodology [55]. By the
end of the pre-production phase a DSL has to be created by modifying the reference
DSL, and it will be used and refined in the production phase in an iterative manner as it
was suggested in [54, 59]. The reuse of previous DSLs can lower the cost of creation
DSLs, and the technologies related to the repository of DSLs still need further
exploration. What I proposed here is a game-centric methodology for creating DSLs, where the game design is the driven-force throughout the language lifecycle.

2.5 Domain Frameworks and Target Environment

Models in MDD are either directly executable, or possible to be transformed into the executable code. In the former case, domain frameworks are the software systems that process the model semantics. In the latter case, as it was described in [24], the models are the software layers that provide the interface between the generated code and the underlying target environment. A game engine is a typical domain framework for MDGD.

The domain frameworks used by the MDGD approaches mainly fall into four categories. The first category is the simplest case where the generated code runs directly on top of the OS or sorts of graphics SDK. The approaches in this category include [RC1], [A1], and [FR1]. The second category is using game engines or equivalent software components as the domain framework. The approaches in this category include [HO1] and [WM1] that used Microsoft XNA (http://www.microsoft.com), [M1] that used Corona SDK (http://coronalabs.com/products/corona-sdk/), and [PH1] that used Adobe Flash (www.adobe.com). The third category is modifying a game engine to promote it to a domain framework as it was suggested in [55], and this category included [F1] that modified an open engine developed by DigiPen Institute of Technology (no longer available online), [F2] that modified FlatRedBall engine (http://newsblog.flatredball.com), and [S1] that added a layer (called “metaframework”) to GTGE engine (http://goldenstudios.or.id/products/GTGE/). The last category is using a specific semantics engine to interpret the models in runtime. [MS1] is the only approach in this category that implemented a full-functioning semantics engine. It can load and execute the model according to a set of semantic rules and render the game graphics as well as handle the player input.

Code generation is the usual solution to support model semantics, which requires a target environment to execute the generated code. The choosing of target environment is dependent on the project context. For simple games such as educational games or prototype development, the graphics API provided by the OS is a proper target environment. On the other hand, the game engine is undoubtedly the mainstream choice for commercial game development, because it is too difficult or even not possible to reimplement the features provided by game engines for a single project. But for most cases, the modification to the target environment (graphics SDK or game engine), such as adding a layer to support generated code is necessary. The modification promoted the target environment to a domain framework, which provides utility code or components to make the generated code simpler [24]. The model interpreter is another solution to support the model semantics. It differs from code generator at two points: analysis time and mode of execution [60]. Interpreters have two distinct advantages: they avoid code generation and compiling thus simplifies the workflow, and they support runtime update of models. Model interpreters have been used successfully in other software domain, such as business process modeling using the Business Process Execution Language (BPEL)[61]. They also seem useful to industrial game development, because compiling and making binary can take much time in a large project.
Based on the above discussion, I suggest that the domain framework for industrialized MDGD should: 1) be based on an existing game engine, and 2) should modify the engine to simplify the generated code. A suitable game engine to fit this purpose is critical for MDGD industrialization, because the features supported impact both the technical quality of games, and gameplay possibilities. It also needs to be carefully evaluated that how easy the engine can be extended to a domain framework. A gap between existing MDGD approaches and the industrial game development is that none of the approaches has chosen a popular, full-scale engine like Unity3D (http://www.unity3d.com), Unreal Development Kit (http://www.unrealengine.com/udk/), Torque (http://www.garagegames.com), etc. as the target environment, whilst few commercial games today is still built on immature engines or graphics SDKs that were used by the MDGD approaches. It is therefore necessary to validate and evaluate the MDGD with modern engines towards MDGD industrialization.

2.6 Modeling Languages

Language provides the abstraction for development and as such is the most visible part for developers[24]. In this section we will discuss the modeling languages used in the MDGD approaches. The discussion is organized according to the properties of the modeling languages, including their language fundamentals, and computational models [24].

2.6.1 Language Fundamentals

The language fundamentals include the syntax, semantics and representation of the language. Syntax specifics the conceptual structure of a language, the constructs of the language, their properties, and their connections [24]; semantics defines the meaning of the constructs, how they are mapped to the problem domain; language representation, also known as the concrete syntax, defines the visualization of the language: the notation that models must follow.

For the syntax aspect, we will look into how the reviewed MDGD approaches define their language constructs. As it was summarized in [24] and [62], there are five methods that are usually used to find the language concepts. They are 1) physical product structure, 2) look and feel of the system, 3) variability space, 4) domain expert concepts, and 5) generation output. Table 2.2 shows methods being used by the MDGD approaches to find the language constructs. For the semantics aspect, we look into the meaning of the constructs, and categorize the languages into 1) the semantics directly maps to the problem domain, and 2) the semantics is from general domain. The former represents the DSLs and the latter represents the GPLs. For the concrete syntax aspect, there are two typical language presentations, which are 1) graphical representation, and 2) textual representation.
Looking into the five sources of language concepts (second column in Table 2.2), the Physical Structure of the product is less relevant to MDGD, because most of the games to be modeled are pure software, which run on non-game-specific platforms, e.g. PC and gaming consoles. So the physical structure of the generic gaming system can hardly provide relevant information to the language that aims at a particular game domain. However, pervasive games [63] as an emerging genre can take Physical Structure as an important source for modeling concepts, since they may use hardware systems that are specific to the games. More detailed discussion about pervasive games modeling has exceeded the scope of the thesis, and more information about the game domain can be found in [63]. None of the MDGD approaches were addressing pervasive games, so it is normal that none of them took Physical Structure as the source of the language concepts. The Look and Feel of the system is about how the system externalizes its states and how users interact with the system. In the computer games domain, it is mainly about the game visualization and gameplay. Unsurprisingly most of the MDGD

<table>
<thead>
<tr>
<th>MGDG Approaches</th>
<th>Language Concepts From</th>
<th>Semantics From</th>
<th>Language Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC1</td>
<td>Look and Feel</td>
<td>General Domain</td>
<td>Graphical</td>
</tr>
<tr>
<td>A1</td>
<td>Look and Feel</td>
<td>Problem Domain</td>
<td>Graphical</td>
</tr>
<tr>
<td>HO1</td>
<td>Look and Feel</td>
<td>Problem Domain</td>
<td>Graphical</td>
</tr>
<tr>
<td>M1</td>
<td>Variability Space</td>
<td>Problem Domain</td>
<td>Graphical</td>
</tr>
<tr>
<td>PH1</td>
<td>Look and Feel</td>
<td>Problem Domain</td>
<td>Graphical</td>
</tr>
<tr>
<td>F1</td>
<td>Look and Feel</td>
<td>Problem Domain</td>
<td>Graphical</td>
</tr>
<tr>
<td>S1</td>
<td>Variability Space</td>
<td>Problem Domain</td>
<td>Textual</td>
</tr>
<tr>
<td>FR1</td>
<td>Look and Feel</td>
<td>Problem Domain</td>
<td>Textual</td>
</tr>
<tr>
<td>MS1</td>
<td>Look and Feel</td>
<td>Problem Domain</td>
<td>Textual</td>
</tr>
<tr>
<td>MV1</td>
<td>Look and Feel</td>
<td>Problem Domain</td>
<td>Graphical</td>
</tr>
<tr>
<td>F2</td>
<td>Look and Feel, Variability Space, Domain Expert, Generation Output</td>
<td>Problem Domain</td>
<td>Graphical</td>
</tr>
<tr>
<td>WM1</td>
<td>Look and Feel, Domain Expert</td>
<td>Problem Domain</td>
<td>Textual</td>
</tr>
</tbody>
</table>
approaches used it as the main source of language concepts, and the language designers usually start the language definition with outlining the “Look and Feel” of the target game domain. [M1] and [S1] went further with using a feature model as a formal method to analyze the variability space of the games to be modeled, which led to a more systematic specification of the domain concepts. Moreover, [WM1] encouraged the participation of domain experts (game designers) in the language design phase, which can provide valuable insights that are necessary for technical people to avoid misunderstandings of the target domain. As it was revealed in [62], the concepts from 2) look and feel of the system, 3) variability space and 4) domain expert concepts, lead to higher abstraction than those originating from 5), so we can say all the MDGD approaches were in the right direction regarding the source of concepts. [F2] is the best approach among them from my point of view, where Fortado et al. proposed a workflow for domain analysis and language design, which takes all the sensible methods including 2) look and feel of the system, 3) variability space, 4) domain expert concepts and 5) generation output. The workflow is the most systematic and comprehensive method for finding the MDGD language constructs.

Third column of Table 2.2 presented the semantic source of modeling languages used by the MDGD approaches. The semantics of DSLs to some extent come directly from the problem domain, while the semantics of GPLs do not map to a particular problem domain[24]. Most of the reviewed MDGD approaches (11 out of 12) used Domain Specific Languages, while only RC1 used General-Purpose Language (UML in this case) for modeling the target games. For those languages whose semantics come from the problem domains, the gaps between the language semantics and the meaning of the target domain concepts are reduced but still remain. Basically the higher the abstraction level of the modeling language, the narrower is the gap between the language semantics and the meaning of the domain concepts.

The concrete syntax i.e. language representation determines the first-sight impression to the language. “A poorly chosen concrete syntax will drive users away, stopping them from using even the most wonderful language”. [59] The most commonly used forms of concrete syntax are graphical form and textual form. Graphical representation is usually more intuitive, and by using graphical representation, the language notation and its symbol can be directly taken from the representation of domain concepts, which is the best practice for selecting symbols [24]. As it is illustrated in Figure 2.3, the same syntax and semantics represented by Figure 2.3a and 2.3b have different usability to the general audience.
Figure 2.3 Graphical Concrete Syntax V.S. Textual Concrete Syntax

However, graphical languages does not always over-performs textual languages, as it was discussed in [64]. Another study also revealed that higher proportion of developers might be predisposed to choose text given its traditional prevalence in programming [59]. To take the advantage of both forms, Engelen and Brand explored the potentials of integrating graphical and textual representations and their results were reported in [65], which I think is beneficial for designing game DSLs as well. From the rightmost column of Table 2.2, we can find that all MDSD approaches chose one of the two forms for language presentation, where 8 out of 12 were graphical languages, and 4 out of 12 were textual languages. It is worth noting that there are other forms for language presentation such as matrix, table, trees, and etc. Sometime they are better than graphics or text. For example, a matrix is especially good if connections between model elements are important, and a matrix also scales better than a diagram since more information can be show in the same space [24]. I believe these third-forms of language representation have their potentials in DSL design, and the synergy of combining different forms of language representation in MDGD also needs further exploration.

2.6.2 Model of Computation

Mainstream programming languages follow the imperative computational model that defines computation through a sequence of steps [8]. DSLs can support alternative computational models such as Decision Table, State Machine, and Dependency Network. Alternative computational models are one of the compelling reasons for using a DSL[8]. Modeling languages can be roughly divided into languages that: 1) model static structures, and 2) specify dynamic behavior [24]. Both kinds of languages were found in the literature review, and we classify the languages according to their model of computation in table 2.3. In [24], it is pointed out that the computational model of a DSL can be 1) Structural, 2) Behavioral, and 3) Hybrid. Most of the modeling languages in Table 2.3 belong to the hybrid model, and I will discuss both the structural and behavioral aspects of each language.
### Table 2.3 Computational Models of Modeling Languages

<table>
<thead>
<tr>
<th>MDGD Approaches</th>
<th>Computational Model of Language</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Structural Model</td>
</tr>
<tr>
<td>HO1</td>
<td>Static Structure</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FR1</td>
<td>User Interface</td>
</tr>
<tr>
<td>MS1</td>
<td>Game World</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>Game World</td>
</tr>
<tr>
<td>M1</td>
<td>Game World</td>
</tr>
<tr>
<td>MV1</td>
<td>Game World</td>
</tr>
<tr>
<td>WM1</td>
<td>Game World</td>
</tr>
<tr>
<td>PH1</td>
<td>Static Structure User Interface</td>
</tr>
<tr>
<td>RC1</td>
<td>Static Structure</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>Static Structure</td>
</tr>
<tr>
<td>F1</td>
<td>Game World</td>
</tr>
</tbody>
</table>

All languages more or less have the structural aspects in their computational model, and three specific structural models can be identified in the languages: 1) Game World, 2) Static Structure and 3) User Interface. Game World model is included in [MS1], [S1], [M1], [MV1], [WM1] and [F1], which is about what elements the game world consist of, and how the elements relate to each other. A typical example of Game World model was presented in [F1], where the room model created with Room Designer specifies the layout of the game scene (room) by a tile map, and entities (game objects) can be put into the room. The room model defined not only the logical structure and the visual presentation of the game world, but also the spatial relationship of the game objects. Another example presented in [MV1], is similar to [F1], where a level editor is created and used to visually model the game world consisted of game objects such as PacMan, ghosts, walls, and empty cells. The game world models in these two approaches are close to the level data (model) created by game engine tools. Figure 2.4 shows the screenshots of two popular game engine tools for comparison. Apart from the visual modeling, textual Game World modeling is also used in some MDGD approaches. For example, [S1] modeled PacMan as well, but differently from [MV1]. The PacMan and ghosts were not visually put into a graphical scene, but instead they were defined with XML code, where each game object is defined in a section identified by a XML tag, which determined the type of the game object (a wall, a PacMan, etc). The spatial information of the game objects was also specified in the XML code with values such as “X_PACMAN_INIT”, and “Y_PACMAN_INIT”. If we compare the textual modeling, and graphical modeling of game world, we can say the graphical models are easier to create, understand and maintain, especially when the model scales with the complexity of the game. The most common practice for commercial game development today, is to create and design the game world using visual tools.
The static structure model is the model defining the constructs used in the Game World model, i.e., it is the meta-model of the Game World model. It corresponds to the M1 level, Class Diagram in the UML [66] meta levels. In the MDGD approaches, the static structure model usually specifies the attributes of the game objects and the operations they support. All of the four approaches included the static structure modeling used a UML-like language. They are either direct adaptation of the UML class diagram [66], or a UML class diagram dialect with customized extensions. User interface modeling consists of describing a user interface and aspects involved (e.g., task, domain, context of use) in models from which a final interface is produced [67]. User interface in games are called “Head-Up Display (HUD)”, which are usually implemented as an individual layer, floating on top of the rendered game world. If we relate the user interface model with the UML meta-levels, it is also on the M0 level as the game world model, but they are different in behavioral aspects.

The behavioral aspects of the system tended to be more difficult than the structural aspects [24], meaning that the behavioral models might be more valuable in game development. For the behavioral computational model, more variations were identified in the MDGD approaches. Script is the method on the lowest level of abstraction, which introduces a 3GP language following the imperative model [8] into the modeling process for describing the game behavior, and it is usually used as a complementary of other kinds computational models. GML2D [HO1] is the language incorporating script with state machine to describe the game behavior, where the game logic that can’t be modeled with state machine can be described with so called ActionScript. ActionScript is not a specific language, instead it can be any language that the target platform supports. State Machine was used by [HO1], [PH1] and [RC1] for modeling the game behavior. A state machine divided the object into a set of states and triggering behavior with events so that each event leads to a transition to another state depending on the current state of the object [8]. Games are suitable for state machine modeling, due to the state-based nature of objects (entities), which react to and fire events according to their

Figure 2.4 Screenshot of Two Level Editors From Commercial Game Engines

(a) Level Editor of Torque 2D (b) Level Editor of Unity
states, and play a central role in the game architecture. State machine as a design pattern [68] has been and still is popular in game development, and the scripting languages by game engines such as Unreal Script have provided built-in support for the state machine. In MDGD, state machines can potentially be more intuitive and easier to use than engine scripting, due to the use of graphical notation. The notation for state machine in [PH1] and [RC1] were quite similar (if not the same) to the UML activity diagram (the notation in [HO1] was not presented with sufficient details). Production Rule System is another important computational model used by the MDGD approaches, which implements the notion of a set of rules, where each rule has a condition and a consequential action, and the system runs the rules through cycles, and executes the rules’ actions whose conditions match [8]. The Rule is an important concept in the domain of game design, so using the rule-centered production rule system as the computational model reduced the problem-solution gap. [RC1] used Production Rule System for modeling the game control, which is simply mapping each input event to a character action, and when an event is fired (condition), e.g. the player pressed a specific key on the keyboard, and the player character will perform the consequent action. The Production Rule System in [F1] was used for more general purposes: the language defined multiple types of conditions including the input event, and the actions to be triggered were not only restricted to character actions but extended to various game state manipulations such as spawning game objects, playing animations, and etc. Moreover, the conditions can be the combination of multiple atomic conditions in conjunction, so are the actions. For instance, it is possible to define a complex rule like: IF <Space key is pressed> AND <the player character has a weapon>, THEN <spawn a Fireball> AND <play fire animation on the player character>. An important feature of the Production Rule System is chaining, which means when the rule actions changed the state of the software, all the rule conditions need to be evaluated to see if any of them have become true, in which case they get added to the agenda [8]. Rule chaining is supported by [FR1] and [MS1]. [FR1] explicitly defined the ”Action Chain” construct in its modeling language, which consists of at least one action trigger (condition) and one action result. More than one triggers (conditions) can be combined in an Action Chain to describe the requisition of an ordered sequence of actions from the player, or an arbitrary sequence of actions. In the former case, the Action Chain implemented the chaining feature of the production Rule System. In the latter case, it implements the conjunction of atomic conditions, which is similar to [F1]. [MS1] implemented an implicit form of rule chaining, where the condition of rule was represented by a combination of flags with Boolean values. If the whole condition is evaluated as true, consequent actions will be performed, which can for example be consumption of an object, triggering a cut-scene, etc. An important alternative of the actions is activation of a flag, which has the potential of triggering another action, since the flag could be (part of) the condition of that action. Thus the rule chain is built implicitly in terms of the correlated flags and actions. Furthermore, the combination of the flags in a condition could be either conjunctive and disjunctive, where the latter as a valuable feature was not supported by other MDGD languages. It is worth pointed out that Production Rule System still has other concerns, such as the contradictory inferences [8] that are important but were not addressed by the existing MDGD approaches, so further explorations on applying the computational model in MDGD are still needed. Decision Trees [69] as a popular technique in the information theory field was also used in the
two MDGD languages, [MS1] and [WM1] both used the Decision Tree as the computational model for modeling conversations between the player and the NPC. The conversation is a tool of game design for unfolding the story, or for determining the game flow. The simplest form of conversations only includes a sequence of sentences spoken by the player and the NPC alternately such as "How are you?" "Good, and you?" "I am good too!". The normal form of conversations in games may be interactive, meaning that the player can choose several alternatives of response to the speech of NPC, where each alternative may lead to a different response from the NPC consequently, thus different information may be delivered to the player and even the global game flow can be affected. The interactive conversation is intuitively suitable to be modeled as a tree with a trunk and branches, and both the MDGD approaches used textual notation to model the tree. Decision trees may also be used in modeling game flow elements other than the conversations, if the elements have a tree-like nature. For example, the level transitions: in the well-known action game Super Mario Bros (Nintendo 1986), you will go to World (level) 1-2 after finishing World 1-1, but from World 1-2 you can not only go to World 1-3, but also go to World 2-1, 3-1 and 4-1 if you find a secret zone. The level transition here can be modeled as Figure 2.5(b) (transition conditions are omitted).

![Figure 2.5](image-url)  
(a) Secret Zone in World 2-1 and (b) Level Transition Tree (Relevant Part) of Super Mario Bros

Such applications of decision trees as shown in Figure 2.5 were not presented in the current MDGD approaches, but I believe it is worth further research.

### 2.6 MDGD Tooling

Model-Driven Software Development does not make sense without tool support [60]. Kent also advocated in [29] that “Tooling is essential to maximize the benefits of having models, and to minimize the effort required to maintain them.” In this section I
review the tool support provided by the existing MDGD approaches, and the development environments for creating them. Then I discuss the gaps between the approaches and the actual situation of the game industry.

2.6.1 Tooling of the MDGD Approaches

The tool support of the MDGD approaches is varied: the simple solutions only provided a code generator while the complex solutions additionally provided a model editor and even a semantic validator. The meta-tools (tools for creating MDGD tools) are also different from one approach to another, which included EMF-based tools, Microsoft DSL tools and so forth. Table 2.4 summarizes the tool support of the MDGD approaches and the corresponding meta-tools used.

<table>
<thead>
<tr>
<th>MDGD Approach</th>
<th>Meta-Tools</th>
<th>Tool Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>[FR1]</td>
<td>Xtext</td>
<td>Model Editor</td>
</tr>
<tr>
<td></td>
<td>Xtend</td>
<td>Code Generator</td>
</tr>
<tr>
<td>[F1]</td>
<td>Microsoft Visual Studio DSL Tools</td>
<td>Model Editor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code Generator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Semantic Validator</td>
</tr>
<tr>
<td>[S1]</td>
<td>Extensible Style Sheet Transformation (XSLT)</td>
<td>Code Generator</td>
</tr>
<tr>
<td>[A1]</td>
<td>Xpand</td>
<td>Code Generator</td>
</tr>
<tr>
<td>[HO1]</td>
<td>Microsoft Visual Studio DSL Tools</td>
<td>Model Editor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code Generator</td>
</tr>
<tr>
<td>[RC1]</td>
<td>MOFScript</td>
<td>Code Generator</td>
</tr>
<tr>
<td>[M1]</td>
<td>GMF / EuGENiA ATL Xpand</td>
<td>Model Editor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code Generator</td>
</tr>
<tr>
<td>[MV1]</td>
<td>DiaMeta</td>
<td>Model Editor</td>
</tr>
<tr>
<td>[MS1]</td>
<td>N/A</td>
<td>Model Execution Engine</td>
</tr>
<tr>
<td>[WM1]</td>
<td>Xtext Xpand</td>
<td>Model Editor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code Generator</td>
</tr>
</tbody>
</table>

2.6.1.1 Tools Provided By The MDGD Approaches

In MDD, models must be executable, either directly with an interpreter or indirectly with a code generator. Table 2.4 shows that 11 out of 12 MDGD approaches support a model execution mechanism. Although [MV1] did not provide the detailed information about its model execution, it does not necessarily mean the mechanism was actually missing. Code generator is arguably the most widely used way of doing MDD [60].

37
reviewed MDGD approaches are in line with the argument that at least 10 out of 12 approaches provided a kind of code generator for transforming the models into the executable code. Various code generation technologies exists, and they can be categorized in different ways e.g. [60, 70]. If we take the classification suggested by [60], the MDGD approaches mainly used two different techniques, which are Templates and Filtering, and Templates and Metamodel. The first technique uses templates to iterate over the relevant parts of a textually represented model, and generates code in terms of the templates whose variables are bound to values from the model. The technique is straightforward to implement but has significant limitations that it is 1) dependent on the concrete syntax of the model instead of the abstract syntax that is more stable during modeling language evaluation, and 2) unaware of the semantics of the modeling language that makes it impossible to do the semantic verification of the model. Due to these limitations, all MDGD approaches except [S1] have chosen to use the other technique, Templates and Metamodel. The technique uses a multi-stage generator that first parses the textual representation of the model, then instantiates a meta-model, and finally uses it together with the templates for generation. This technique avoids the above-mentioned disadvantages of the Templates and Filtering technique, and as pointed out in [60] it is especially important in the context of MDD. The interpreter is another mechanism supporting the model-execution, which shares the same underlying principles as the code generation [60]. Only [MS1] in the reviewed MDGD approaches has chose to implement an interpreter. A possible reason is that interpreters are particularly appropriate for supporting behavioral models: the structural aspects are similar among the modeled games, and the models created with the approach are quite behavior-intensive. This is also in line with the argument made in [60]: “It is more common to use generators for structural aspects of a system and interpreters for behavior.”

The model editor is the central tool in MDD [60], however, not all of the MDGD approaches provide a model editor. [RC1] used UML as the modeling language whose models can be created with general UML tools, meaning that a specific model editor is not necessary. The rest of the approaches provide textual languages whose models can be created with general text editors, however language-specific supports such as keyword highlighting, syntax checking are thus missing. Apart from the above-mentioned approaches, there are six MDGD approaches that provide their specific model editors, and can be classified as below, according to their focus of modeling:

1) The class editor; The editor is used to create models on the similar abstraction level as the UML class diagram, whose main elements are classes describing the static structures of the game objects. The MDGD approaches providing this kind of model editors include [HO1]

2) The game world editor: The editor is similar to the level editors from game engines in functionalities, which supports creating the game world consisting of game objects. The editor can also support the manipulation of the attributes of the game objects, including their location, orientation, graphics and so on. The MDGD approaches providing this kind of model editors include [FR1], [F1], [M1], [MV1], and [WM1].
The behavior editor: The editor is used to create the models describing the behavioral aspects of the games to be developed. The MDGD approaches providing this kind of model editors include [FR1], [F1], [HO1] and [WM1].

Before the code generation, the correctness of models must be checked with respect to the meta-model, and this checking is in most cases conducted with a separate tool in order to avoid the unnecessary complication of the code generation [60]. [F1] provides such a tool, which is called Semantic Validator in the article. The semantic rules used to check the models were implemented programmatically and integrated into the tooling environment. Although the tool has a significant limitation that the semantic rules were not part of the meta-model, it is still the only model check tool in all the reviewed MDGD approaches.

2.6.1.2 Meta-Tools
The above-discussed tools were not created from scratch, but instead various meta-tools were used to build them. The simplest use of meta-tools was presented in [S1], which used XSLT [71] to write the specification of the code generation. The XSLT document can be executed by any XSLT execution tool (Microsoft Visual Studio for example) and generate the target code. XSLT is a language for transforming XML documents into other XML documents. But it is also possible to transform XML documents into documents in other languages such as GPLs. The transformation is based on the concrete syntax of the modeling language (XML in this case) instead of a meta-model that formally defines the abstract syntax and static semantics. Because the concrete syntax is usually more complex than the abstract syntax, the scalability of the approach may be a problem. Also if the concrete syntax changes (i.e. from a textual to a graphical one), the code generation has to be re-created.

More powerful MDGD tools require more advanced meta-tools. Eclipse Modeling Framework (EMF) is an open-source framework that many meta-tools are based on. EMF provides eCore, a meta-meta-model for defining DSLs, and a couple of additional generators that generate editors and a generic editing framework for editing the models [60]. MDGD tools created with EMF based meta-tools are integrated into the eclipse platform together with the meta-tools, thus the meta-modeling and modeling share the same environment. According to the classification of tooling approaches in [24], EMF based meta-tools enable the approach with the best maturity that is described as ”Integrated meta-modeling and modeling environment”. ATL [72] is a Domain Specific Language and corresponding toolkit built on EMF for creating model transformations, that is the process for semantically turning the model in one language to another. Code generation can be implemented as a kind of model transformation if we look on the target code as a kind of textual model. Since ATL transformations (code generators) are defined according to the source and target meta model, it is dependent on the abstract syntax of the modeling language, thus can avoid the disadvantages of template-based code generation that we discussed above. [PH1] used ATL to create the code generator for their DSL, which takes the models in the DSL as input and generate Adobe Flash files. Since ATL requires both the source and target meta-models, the authors created a meta model for Adobe Flash content in addition to the meta model of the DSL itself. [PH1] used ATL for the full code generation, while [M1] chose another
method. In [M1], the code generation included two steps. In the first step, the models in game DSL were transformed into intermediate models. In the second step the intermediate models were used to generate the target code. They implemented this two-step generation because the gap between the abstraction levels of the game DSL and the target framework was so wide that a single-step code generation is considered as improper. The second step of the code generation in [M1] uses Xpand to do the model-text transformation. Xpand is actually a complete code generation technology, which conforms to the "Templates and Metamodel" type in Stahl and Völter’s taxonomy [60]. Xpand provides a statically-typed template language specifically for model-text transformation, which takes models conform to any eCore meta-model as input and can generate target code in any language. The Xpand language comes with a set of tools for enabling the language on the Eclipse platform. Xpand was the most popular tool for code generation in the MDGD approaches, e.g. [A1], [M1], and [WM1]. Xtend (http://www.eclipse.org/xtend) is the successor to the Xpand project, which is a statically-typed programming language that translates to comprehensible Java source code. Programming with Xtend is similar to java, however it has some improvements including Template Expressions that makes Xtend suitable for code generation. Comparing to template languages like Xpand, Xtend has some advantages, e.g. since Xtend code will be translated to java before execution. Its runtime performance is as good as java code, which is much faster than typical template languages that are executed by interpreters. Further, Xtend code is debuggable while template code is not. Xtend was used by [FR1] for code generation. The Xtend language is implemented on Eclipse platform and is usually used together with Xtext to enable textual DSM on Eclipse. Xtext (http://www.eclipse.org/Xtext) is an open source framework for developing DSLs on Eclipse platform. The framework provided a language and a user interface for defining the DSLs. Once a DSL is defined, the framework generates a parser, a class model for the abstract syntax tree, and an IDE integrated into the Eclipse platform for it. The IDE is actually a textual DSL model editor offering syntax coloring, code completion, static analysis, and more. Xtext was used by [FR1] and [WM1] for defining the DSLs and creating the model editors. Xtext supports only textual DSLs, whilst GMF and DiaMeta are two alternatives for implementing graphical DSLs on Eclipse/EMF platform. DiaMeta takes an EMF model as the meta-model of the DSL, and an editor specification that is created with DiaMeta tools as the concrete syntax specification. A graphical model editor is then generated. DiaMeta was used in [MV1] to create the model editor. GMF (http://www.eclipse.org/modeling/gmp/) is a project similar to DiaMeta for defining graphical DSLs and creating model editors correspondingly. GMF supports customization of the generated editor with manual code, which makes it more flexible than DiaMeta. GMF was used in [M1] for creating the model editor.

Apart from the open source meta-tools on Eclipse/EMF platform, there are also some closed-source meta-tools such as MetaEdit+, and Microsoft DSL tools. Microsoft DSL Tools is the only one being used by [F1] in the reviewed MDGD approaches, which is a powerful toolkit integrated frameworks, languages, editors, generators, and wizards that allow users to specify their own modeling languages and tools [24]. The toolkit provided its own meta-meta model for defining DSLs, which is equivalent to eCore in EMF. One limitation is that Microsoft DSL Tools is only available on the Windows
platforms as part of Visual Studio. It was used by [F1] to create the tool-suite for a DSL including a model editor, a code generator and a semantic validator.

2.6.2 Challenges with MDGD Tooling

The MDGD approaches have provided different levels of tooling support, from the basic code generator to the integrated tool suite, which makes it possible to develop an actual game with the approaches. However, all of the approaches are results of academic projects, while none of them has been applied in the real commercial game development so far. From my own experiences from game development, I can see the gap between the tools provided by the MDGD approaches and the tools that the industry needs. In this section I will summarize the challenges with applying the MDGD tools in commercial game development.

2.6.2.1 MDGD Tools and Game Engine Tools Interoperability

The game industry has a long history of using game engines including the tool suites coming with the engines. When MDGD is introduced to game production, its tools must be integrated with the engine tools instead of replacing them, and this is because:

1) The engine tool suites are mature software products that possess high stability as well as usability. Embracing such tools instead of ignoring them can save unnecessary effort on developing competitors of the industry-strength tools, whose quality has been proven in long-term development practices.

2) The potential users of the MDGD tools may have valuable experience of the engine tools. Successful commercial engines (e.g. Unreal and Torque) have been used in game industry for many years thus have built up their user community and online knowledge base, with which it is relatively easy to train new developers, or hire developers with the experience of such tools.

3) Some tools are not necessary to be recreated for each MDGD domain. Tools such as the world editor and the resource manager may be reusable among different game domains, i.e. they are relatively general in purpose. For example, Microsoft’s XNA game framework provided a world editor that is generic enough for creating game worlds for various genres of games.

4) The engine-based development methodology has been dominating the industry for years and the game development process has been adapted gradually to fit the situation. It is expensive and unnecessary to renew the process to adopt the MDGD methodology.

All in all, as it was advocated in [73]:

“A prudent and practical way to introduce new technology and techniques into an existing production environment is to apply them to a smaller-scale project such as a relatively low-profile extension to some legacy system. This implies not only that the new software must work within legacy software but also that the development process
and development environment used to produce it must be integrated into the legacy process and legacy development environment.”

However, most of the existing MDGD approaches are only aware of the run-time engine, while none of them provide a solution to integrate the MDGD tools with the engine tool suite. Moreover, in some approaches, tools have been developed to re-implement the functionalities provided by the engine tools. For example, a world editor is the core tool in most (if not all) of the engine tool suites, which was re-implemented in SLGML [F1], AcadEx [F2], and PacMan DSL [MV1]. This reveals a gap between the MDGD approaches and commercial game development due to the interoperability of MDGD tools and the engine tools.

To enable the interoperability of MDGD tools and the engine tools, model transformation has to play a key role. Model transformation is the automated processes that take one or more source models as input and produce one or more target models as output [74]. By model transformation, the models created with MDGD tools can be transformed into the models that can be accessed with the engine tools. In traditional terminology, the latter is called “data” but it is actually a kind of models in an engine-specific format. It is usually not difficult to develop parsers that can deal with the data format to retrieve the semantics, when the data format is sometime just plan binary image of engine objects, and may even be a human-editable format (e.g. level data created with Torque 2D engine). There might be some practical difficulties with the implementation of such model transformation, but in theory no more complicated than the transformation bridging MDGD approaches with multimedia authoring tools such as Adobe Flash, which has been done in [PH1]. Furthermore, sometimes the transformation from MDGD tools models to engine tools models must be reversible to support round-trip engineering. This is necessary to ensure consistency among models when changes are made in one of them, in order to enable a flexible workflow that is highly demanded in an innovation-driven and change-intensive industry.

2.6.2.2 Model-level Debugger

Model-level debugger is the tool that helps modelers to identify semantic errors and locate them at model-level. The static semantics can be checked with the semantic validators as done in [F1], but the dynamic semantic errors have to be identified through observing the models at run-time. Without the model-level debugger, such errors can only be reported from the generated code but cannot be traced back to the model-level. This could bring significant problems to the productivity of MDSD in general, since extra time have to be spent on locating known code-level errors at model level, which can be even more expensive than locating known binary-code level errors at GPL program level. This is because the semantic gap between the abstraction levels of the modeling language and the generated code is wider [73]. More details about the challenge of missing the model-level debugger can be found in [73], where the tool was classified into “model-level observability” tools together with the model mergers and model difference tools. We argue that the model debugger is particularly important for MDGD, because one important purpose (or advantage) of MDGD is to enable game designers and level designers make as much the game logic as possible without going through programmers to save the communication cost and avoid misunderstandings. But
if (dynamic semantic) errors can only be reported at code level, most of this advantage is nullified, as professional programmers are still needed in the modeling process to do the bug-fix, unless the designers themselves are programmers, which is not likely the case in game projects.

2.6.2.3 Cooperation of Model Interpretation and Code Generation

As we have discussed, there are two alternatives for making models work, which are code generation and the interpretation [8]. Interpretation tools allow models to be executed without producing intermediate output. One important advantage of interpretation is that they provide early direct experience with the system being designed [73]. There are at least two reasons that make interpretation tools particularly valuable for game development: One is that the game development suffers a workflow problem, as compiling game code from scratch or when a major C++ header file is changed, it can take half an hour or longer [1]. The other is the modelers in game development sometimes are game designers or level designers who might not be able to use the programmer tools such as compilers and binary-image-making tools. On the other hand, code generation also has its unique advantages for game industry, such as the performance and image size, which are both critical for many games. This means that the integration of model interpretation and code generation is demanded when applying MDGD in commercial game development. However, all MDGD approaches except [MS1] only support code generation, and [MS1] supports only model interpretation, which makes it hard to integrate the interpreter and code generator, leaving a gap between the MDGD approaches and the commercial game development.

General code generation technology can be directly used in game production, but the model interpretation in game development needs further explorations. Interpretation technology has been used to execute programs written in GPLs such as BASIC, java script for long time, and it has also been used to execute domain-specific models such as the BPEL engine for business process models [75]. Most of the game world editors also support interpretation of the world model to provide instant feedback to game designers or level designers. The general requirements to the model interpretation tools were discussed in [73], and here we point out more requirements specific to game domain:

1) **The model interpretation tool must cooperate with the world editor.** When the models are executed, the result must reflect the newest change in the game world model, which was made with the world editor. Because the models created with MDGD tools are usually related with the game world model, without the newest game world model, the models may not work properly. For example, a behavioral model of a specific gameplay object cannot be verified in the real game execution without the object being created in the game world model.

2) **The performance of interpretation must not break the pace of gameplay.** Interpretation is usually less efficient than compilation, so when it is used in performance-sensitive games such as action games (ACT), first-person shooters (FPS), the gameplay may be impaired with the performance, thus the gameplay models can become difficult to verify. For example, with too low frame rate, it can
be difficult to aim your enemy in an FPS, which makes it difficult to verify the behavior. Therefore more effort is required on the optimization of the model interpretation tools in MDGD than in general MDD.

I have discussed the challenges that are particularly important with the existing MDGD approaches. Actually successful MDD may require more tools that are lacking in current MDGD approaches, and such tools are summarized in [76], and include model validation tools, model instances management tools, model mapping tools, model-driven testing tools, dashboard applications, version-control and distributed modeling tools, and software process management tools.

2.7 Game Architectural Concerns

The functionalities and complexities of computer games vary, so the functional and non-functional requirements of different games require various game architectures to be satisfied. A single player game like PacMan may be implemented with a single program, while a Massively Multiplayer Online Game (MMOG) like World of Warcraft (Blizzard 2004) has to be implemented with multiple software systems running on thousands of servers and millions of player machines. The evolution of computer games in last decades led to the evolution of game architectures, and today many game architectures have been developed to support the diversity of games. Walt Scacchi also pointed out in [7] that software architecture is one of the research question and challenge for future game development. In Chapter 4 we will review the most important game architectures in the literature.

A significant gap between the circumstance of commercial games and the MDGD approaches are that all of the approaches only concerned with single software architecture, while none of them addressed the networked game architectures, which are required in most popular commercial games. To make MDGD practical, more architectures need to be supported. In this section I point out two important architectural concerns that will be addressed later on in this thesis.

2.7.1 The Diversity of Game Architectures

Most of the modern games today more or less provide online features. Multiplayer online games such as MMOGs support the massive player interactions through the Internet, and even a single player game on mobile phones may also support light-weight online features such as high-score list and downloadable content. Single software architecture is not sufficient to fulfill such requirements, so more game architectures have to be supported in MDGD, e.g. client-server, peer-to-peer, multiple server, just name a few. Since the requirements vary among games, for a given game domain, different architectures may have different appropriateness. Choosing a proper game architecture should be the first and foremost task for MDGD, because the game architecture influences many aspects of the MDGD. Here I point out three important ones:

1) The DSLs: Architecture-Centric MDSD (AC-MDSD) is an MDD flavour addressing architectural concerns. In general AC-MDSD, the model must contain all relevant
information for the generation of the infrastructure code to support the target architecture [60]. This is also true in the specific game development domain, since game architecture is also a kind of software architecture. So the DSLs of the MDGD approaches must support the specification of the target game architecture at the language level.

2) **The code generation:** Complex architecture usually derives complex infrastructures that developers are confronted with: application servers, data bases, Open Source frameworks, protocols, and so on [60]. Implementing such infrastructures is a tedious and error-prone routine work, which should be automated by the MDGD code generation.

3) **The domain framework:** To enable the model execution on the desired architecture, the domain framework must support the target architecture.

Making a practical decision about a game architecture requires the comprehensive knowledge in the game architecture domain, and an effective decision process. The latter aspect is out of the scope of my thesis. For the former, a thorough investigation of the game architectures is needed, which I will present in Chapter 6.

### 2.7.2 The Proper Abstraction of Game Architectures

In general AC-MDSD, the architectural information contained in the models is a lot more abstract and more compact than the expanded code [60]. This means that the abstraction level of architectural elements in the DSL is a lot higher than the actual architecture. This is particularly important in the MDGD domain, because in game development the modelers who use the DSLs are usually non-programmers (game designers for instance) who possess very little knowledge about the software architecture domain. The MDGD approaches have to raise the abstraction level of game architecture, and leave as much implementation details as possible out to the code generation and the domain framework, in order to properly lower the threshold of applying MDGD. In chapter 4 I will introduce my conceptual framework for game architectures, which serves a base for making the proper abstractions in the game DSLs that support various game architectures.

### 2.8 Summary

Model-Driven Game Development (MDGD) adapts the general Model-Driven Development methodology to game development domain, which can potentially make game development faster and easier. Apart from the advantages of MDD in general software development, MDD is particularly useful for game development, because a major part of game developers are non-programmers, so use of models instead of code can not only make their life easier but also save cost in communication and prototyping. But as a new methodology, MDGD still has not been widely accepted by the game industry, and to my best knowledge no commercial MDGD project has been reported so far. In this chapter I reviewed 12 representative academic MDGD articles in the literature, and discussed the gaps between the academic studies and the industry circumstances.
The reviewed MDGD approaches aimed at different game domains, whose scopes were quite different. The broad scopes covered wide range of games such as all 2D games, while the narrow ones covered only one specific game. The domain specifications are also in different level of details. Based on their work, I argue that a proper domain definition for MDGD should include both the horizontal domain and the vertical domain, whose specifications should be formal. To this end, systematic frameworks for characterizing game domains are demanded, which is a future research direction in the field. Moreover, I argue that the horizontal scope of MDGD domain should be narrowed down to a specific game. To amortize the cost of building the MDGD artifacts, the DSL and the corresponding tools should be achieved in a repository together with their domain specifications for reuse with modifications in future projects.

The domain frameworks chosen by the MDGD approaches also vary. Domain framework is the software that implements the domain semantics thus makes the models executable. If code generation is used in a MDGD approach, the software underlying the generated code is the domain framework, and if model interpretation is used, the interpreter is the domain framework. In the simple cases, some MDGD approaches use graphics SDK or OS as the domain framework and generate code running on top of it. Game engines are on a higher abstraction level than graphics SDK and OS, which are used by some of the MDGD approaches as the domain framework. Moreover, there are also some of the MDGD approaches that choose to modify existing game engines to better support the generated code. Model interpreter has also been used in domain frameworks, but only in 1 out of 12 reviewed approaches. None of the MDGD approaches chose a full-scale commercial engine as the domain framework, while prototypes on immature engines or graphical SDKs still cannot sufficiently justify the usefulness of MDGD in commercial game development, which are mostly based on commercial engines. To validate MDGD on commercial engines is a future research direction.

The modeling languages used by the MDGD approaches were also discussed in this chapter from the language fundamentals and computational models perspectives. Most of the approaches have chosen DSLs instead of GPLs. Most of the approaches have looked for the constructs of modeling language from only one or two sources, apart from [F2], which proposed a more insightful perspective including four sources. The languages used in the MDGD approaches are either graphical or textual, while other forms such as matrix and tree are overlooked, which I believe worth further research. Regarding the computational models, there are two categories: the structural model and the behavioral model. The structural computational models supported by the MDGD approaches included game world, static structure, and user interface. The behavioral computational models supported by the MDGD approaches included decision tree, script, state machine, and production rule system.

Tooling is essential for MDD to become useful. Tools provided by the MDGD approaches and meta-tools for creating the tools were also discussed in this chapter. There are four kinds of tools presented by the MDGD approaches: code generator, model editor, semantic validator, and model interpreter. Most of the MDGD approaches have chosen Eclipse/EMF based meta-tools, while Microsoft DSL tools and MetaEdit+
were also used. The gaps remaining between the MDGD approaches and the commercial game development were summarized as 1) the interoperability between the MDGD tools and engine tools, 2) model-level debugger, and 3) cooperation of code generation and model execution.

The architectural diversity of modern games has been overlooked by the existing MDGD approaches: None of the approaches have been validated in a networked-game setting, in spite of the popularity of today’s online games. Choosing a proper game architecture is the first and foremost step of MDGD, since the game architecture influences the DSLs, the code generation, and the domain frameworks. How to reflect the architectural characteristics in the models is an interesting research question, which requires a proper abstraction of game architectures. A conceptual model of game architecture can be helpful in making the abstraction, and I will present such a conceptual model in Chapter 5.
3 Engine- Cooperative Game Modeling (ECGM): A Hybrid Approach Bridging MDGD Tools and Game Engine Tools

Game engines are today used frequently in game development, which lowers the threshold of game production by implementing common technologies and providing convenient content-creation tools. As we have discussed in Chapter 2, some of the existing MDGD approaches have been aware of the importance of game engines in industrial game development, thus run-time game engines were used as a part of the domain framework, or the base of the domain framework in the MDGD approaches. However the other parts of the game engine like the engine tool suite were still overlooked, which leaves a gap between the current MDGD approaches and the commercial game development.

In this chapter I will look into the problem and propose a hybrid approach named ECGM to address the full integration of MDD tools and game engines. The approach not only makes use of the run-time engine, but also makes the MDGD tools and engine tools interoperable by utilizing model transformation or code generation technology to transform MDGD models to the content data that engine tools support. ECGM minimized the change to the existing workflow and technology, thus reduced the cost and risk of adopting MDGD in commercial game development. This is therefore one important step towards industrialization of MDGD

3.1 Game Engine: The Past and The Present

The term “game engine” arose in the mid-1990s in reference to first-person shooter (FPS) games like Doom (id software, 1993) [9]. This new genre introduced realistic 3D real-time rendering, which required more complex programming than 2D games. It became very expensive to re-implement all the 3D rendering functionalities for each FPS game, so the software reuse caught the attention of the game industry. Doom was
architected with a well-defined separation between its core software components (such as the three dimensional graphics rendering system, the collision detection system, or the audio system) and the art assets and gameplay, enabling reuse when developing new products by creating new art, world layouts, and other gameplay elements [9]. However, at the stage of Doom, 3D was still the exception rather than the rule. Most games were still being developed in 2D, until Quake (id software, 1996) heralded in the new 3D era of games [77]. Quake and its successor Quake III Arena (id software, 1999) not only pioneered the realistic 3D rendering, but also led the industry to the new era of game engines. Quake III was designed with reuse in mind, which were made highly customizable via a scripting language named Quake C [9]. The game world can be created with a tool named Q3Radiant, and saved in a specific data format that can be loaded and rendered by a run-time engine officially named ID Tech 3. The reusability of ID Tech 3 was proved over years by at least 75 games¹ that were developed using the same engine, and the most important ones include Call of Duty 2 (Infinity Ward, 2005), Star Trek: Elite Force II (Ritual Entertainment, 2003), just name a few. After several years of proprietary license, it was announced that the engine was now released under GNU GPL (version 2) in 2005. As the major competitors of Quake series, Unreal series including Unreal (Epic MegaGames and Digital Extremes, 1998), Unreal II: Awakening (Legend Entertainment, 2003), Unreal Tournament 3 (Epic Games, 2007) were also FPS games featured reusable 3D game engines. The 3D game engines powering Unreal series were named Unreal Game Engine including Unreal, Unreal 2, Unreal 3, and most recently Unreal 4. Apart from some unique 3D features, Unreal engines provided a domain-specific scripting language named Unreal Script that raised the abstraction level of game programming a bit to the problem domain, and a powerful visual tool for creating game world named UnrealEd. Unreal Game Engines were so successful that there are over 100 games² released spanning from 2003 to 2014 based on Unreal 2, and there are even more for Unreal 3³. Game engines are typically somewhat genre specific (FPS for example), however it is also possible to construct games of other genres with a specific engine [9]. The difficulty varies when engines are reused in genres other than their original genres. Unity Engine (Unity Technologies) is a game engine that supports several genres from 3D to 2D games. It also supports a wide range of target platforms including desktop computers, gaming consoles, handheld consoles, tablets and smart phones. The reusability of Unity Engine is so high that there are also many non-gaming projects built on the engine, for example [78, 79]. Research of John Austin found that the cost of developing a proprietary 3D engine for a game takes from between 40 to 70% of the total development budget [77]. Therefore reuse of exiting mature technology can significantly save the cost of production. Due to the financial benefits, game engines have changed the game industry, and today engine-based development has become the mainstream approach for the production of commercial games.

I have presented a brief history of the game engine, and pointed out some important engines as the landmarks alone the road of the engine evolution. However, if we ask “What exactly is an Engine”? It is still not easy to come up with an answer. There are

¹ http://www.uvlist.net/groups/info/idtech3
fundamental differences between the exiting definitions in literature [80]. In [9], a game engine was defined as “software that is extensible and can be used as the foundation for many different games without major modification”. The definition clarified that the engine is not the game itself but the foundation of the game, and emphasizes its reusability and extensibility. However, the author also pointed out that the line between a game and its engine is often blurry, and some engines make almost no attempt to separate the two [9]. In [81], the authors further clarified the scope of the game engine in a game by defining the game engine as “… that collection of modules of simulation code that do not directly specify the game’s behavior (game logic) or game’s environment (level data).” Here the line separating the game and the engine lies under the modules simulating the game logic and game environment. Furthermore, since game engines today are also used in areas other than the computer game, in [82] game engine was defined as “the core software component for any applications that uses real-time graphics displays such as electronic video games (computers and consoles) and other interactive applications that contain 2D or 3D graphics rendering”. Here the application domain of game engines is extended to interactive applications with the demands to graphics rendering. The graphics rendering was regarded as the central functionality of the game engine in the definition, which was also confirmed in [81] with the statement that “the rendering engine is the crown jewel of the game”. Apart from the run-time components, people are aware of that game engines are usually released with a tool suite for creating game data. Therefore a question was raised in [81] that if the tool suite should be considered within the scope of the game engine? In [83], a positive answer was provided with the definition that the game engine is “a framework comprised of a collection of different tools, utilities, and interfaces that hide the low-level details of the various tasks that make up a video game”, and it was also supported by the statement in [9] that “a game engine consists of a tool suite and a runtime component.” The opinion is also in line with my industrial experience about how game developers understand the game engine. Moreover, game engines enable a data-driven architecture of game, which is what differentiates a game engine from a piece of software that is a game but not an engine [9]. [84] suggests a definition of a game engine that particularly focuses on the importance of the data-centric architecture: “An engine has to manage all data in its responsibility area; Compute all data according to its task area; Pass on all data to its following instance, if necessary; Accept data to manage and to compute from preceding instances.”

In accordance with the above discussion, when I talk about a game engine in this thesis, I refer to the software kit that includes 1) a run-time system that implements the common technologies for a game domain without the awareness of the game-specific logics, and 2) a tool suite for managing the data to be processed by the run-time system. Correspondingly I use “run-time engine” to denote the run-time system and “engine tools” to denote the tool suite respectively.

3.2 Modern Game Engine Architecture

Game engines today are complex software with tens of modules implementing various functionalities. The concrete architectures can be different for different engines, however most of the modern game engines shared a common architectural framework, which is illustrated in Figure 3.1. Similar figures or discussions with different level of
details about general engine architectures or architectures of particular engines can also be found in the literature, e.g. [9, 81, 85].

**Figure 3.1 General Architecture of Modern Game Engines**

In Figure 3.1, the run-time engine constitutes the mid-ware layer, which lies between the platform layer and the application layer from the layered system [86] perspective. The major components of the run-time engine include graphics renderer, sound renderer, collision and physics system, animation system, scripting system, scene management, resource management, network system, and general AI system.

The **graphics renderer** may be the most important and complex component for a game engine, especially for a 3D engine. It is in charge of the on-screen visualization of the in-memory game world. The major techniques implemented by a 3D graphics renderer include but are not limited to: selecting visible objects, culling invisible surfaces, rendering visible surfaces according to the current camera settings, simulation of lights and shadows, simulating 2D graphical effects, and rendering the heads-up display (HUD). There are a lot to say about the graphics renderer, however, this topic is out of the scope of the thesis. The **sound renderer** is relatively simple, which is in charge of simulating sound effects in a 3D or 2D virtual environment as well as playing the background music. The graphics and sound renderers make objects perceivable, while the **physics system** makes objects interactive. The fundamental part of the physics system is the collision sub-system, which focuses on two problems: the collision detection that is about if there are spatial overlaps between objects; and the collision handling that is about what will happen if objects collide with each other. Usually the former is a geometric problem, while the latter is a physical or AI problem. Some physics systems today also support the rigid body dynamics, which simulates the
motion of rigid bodies, the forces and torques, making the kinematics of objects more natural. The animation system is in charge of the visual simulation of the characters’ motion, including the human characters, non-human characters and even non-organic characters (e.g. robots, machines). Sprite animation and skeletal animation are the most common techniques implemented by the modern engines. Sprite animation is a method that simulates motion pictures by switching a couple of images alternately in the specified time intervals, and it is the main technique for 2D animation and also an important supplementary for 3D animation. Skeletal animation is the main method for realizing 3D animation, which uses a couple of correlated key vertices (called “bones”) to pose the character, and compute the coordinates of all vertices of the 3D model (called “mesh”) according to their position relative to the coordinates of the bones. A skeletal animation includes multiple “key frames” that record the coordinates of bones at designated “key” moments. The coordinates of the bones between key frames are interpolated in terms of the coordinates in the key frames. More animation techniques used in the modern engines include vertex animation, inverse and so on.

The AI and gameplay system is what makes the characters live. AI and gameplay system in a game can be divided into two parts: the game-specific part and the general AI part. The general AI system implements the low level algorithms and mechanics that are essential for creating game-specific AI, such as path-finding, and event management. Since these techniques are common for different games, they can be built as part of the run-time engine or a reusable library. The game-specific AI has to be created for each game, which is usually coded in a scripting language. The execution of the scripting language is supported by the scripting system, which can either be general-purpose languages such as C# in Unity, or domain-specific languages such as unreal script in Unreal. The scripting language is used because such languages make it easier to develop the game-specific AI than the engine-native languages such as C++.

Multiplayer games over network are more and more popular today, which require the network system in the run-time engine to support the interactions between distributed players. The network system can coordinate the clients and servers, replicate game states, address security concerns, etc. Multiplayer mode is so important that many engines actually treat single-player mode as a special case of a multiplayer game [9], therefore the network system becomes an essential part of the modern engine. The scene management system provides common services to many engine components, which organize the objects in the scene with a “scene graph” data structure. The scene graph can for example be a Binary Space Partitioning (BSP), or an Octree, which provides services of fast query and management of game objects. The services are consumed by other systems such as renderer, and AI. Moreover, a complete game includes many kinds of assets, such as level data, script code, graphics and sound data. All the assets require a common interface to load and manipulate at run-time. The resource management is responsible for the job. The assets may be packed into big files for some engines, and the resource management also has to extract the big file in such cases.

I have presented the major components of the modern game engine, and it has to be pointed out that there are many other components that are potentially included in the modern engine architecture but are omitted in Figure 3.1, such as human interface
devices (HID) support and GUI systems. Furthermore, not all the components in Figure 3.1 are necessarily provided in an engine kit, and third-party software is usually the alternative solution in such cases. Therefore integrating one or more third-party software with the game engine is a common task in the game development process.

The layer underlying the run-time engine in Figure 3.1 is what I call platform layer. The layer consists of the OS and some other libraries that implement low-level technologies supporting the run-time engine. The graphics library may be the most important one, which implements the 3D rendering pipeline utilizing the power of graphics hardware. The graphics renderer in the upper layer feeds the graphics library at run-time with triangles, textures, and shader programs (the program manipulating the graphics hardware to typically do advanced rendering job) to get the desired results on the screen. DirectGraphics and OpenGL are two examples of such graphics libraries. Other libraries in the platform layer can be encoders/ decoders, network protocols, font handlers, data structures and algorithms, just name a few.

The layer above the run-time engine is the application layer. Mainly two kinds of applications exist: the game built on the engine, and the world editor that is part of the engine kit. Making the world editor an application of the run-time engine is a recent trend. The old engines such as Quake built the world editor directly on the platform layer, or based it on a thin layer of the run-time engine, i.e. engine core. But the modern engines such as Unreal 2, Torque 2D, and Unity follow Figure 3.1. As it was discussed in [9], this architecture has some significant advantages such as avoiding two representations of data structures between the run-time engine and the tools, fasten the execution of game within the editor, and live in-game editing is easier to support. One more important profit is that what you see in the world editor can be what you get in the live game, because the rendering in the editor and the game is done by the same routine. In [9], a disadvantage of the architecture was also pointed out that when the engine is crashing the tools become unusable as well. This disadvantage cannot be overlooked and I did see this happened as a professional game developer, and it caused significant workflow problems.

It is also worth noticing that for a specific engine, there may exist more tools other than the world editor, such as the plug-ins for digital content creation (DCC) tools[9], packaging and binary tools, profiling and debugging tools, and script development tools. Since they are optional, and their architectures are diverse in different engines, I did not include them in Figure 3.1.

### 3.3 Some Game Engines In Market

In this section, we take a quick look at some representative game engines in the market. I will only introduce the full-scale engines that have all the essential run-time features and sufficient tool support.

#### 3.3.1 Unreal Engines

The Unreal Engines by Epic Games Inc is one of the most popular game engines in the market, on which hundreds of commercial games on computers, consoles and handheld
devices have been developed. Unreal Engines today have released four major versions (Unreal 4 the latest), but Unreal 2 is still widely used in developing modern games, which proved the reusability of the engine. Due to the FPS nature of its native game, Unreal Engines are mostly suitable for developing FPS games and third-person shooters, however, there are still games in other genres built on them in cost of modifying the engine, which is not too difficult to do. Apart from the extensive graphics features, Unreal engines have two significant advantages overcoming most of its competitors:

1) Powerful and domain-specific scripting. Unreal Script coming with the Unreal Engines is a java-like scripting language with most of the OO features. Moreover, Unreal Script provides language-level support for state machine that is particularly suitable for implement game behaviors. This feature raised the abstraction level of the language, although the target domain is still too broad and the domain expressiveness is limited. Since Unreal 3, a graphical scripting language, Unreal Kismet was introduced, which further lowered the threshold of gameplay programming compared with the textural scripting. Unreal Kismet still suffered the relatively broad domain and poor expressiveness, however it was one more step in the right direction to the model-driven game development.

2) Powerful toolset. Unreal Engines provided cohesive and comprehensive tools such as world editor, resource manager, graphical script editor that are integrated into UnrealEd, an IDE for game designers. Due to its impressive functionality and usability, countless mods were developed on Unreal Engines, including many non-game projects, such as [87, 88].

The Unreal Developer Network (UDN) is one of the most mature online knowledge bases for game developers, which is supported by Epic Games Inc. and is available for Unreal Engines subscribers. With UDN, game developers can easily get familiar with and even be proficient in the Unreal Engines.

### 3.3.2 Unity Engine

Unity Engine is developed by a Denmark company named Unity Technologies. The engine offers extensive features that are comparable to other main-steam 3D engines. In [78], the advantages of using Unity Engine in developing a non-game project named SARGE were summarized as below, which were also valid in general projects:

1) Good documentation.
2) Mature online developer community.
3) High usability of the world editor.
4) State-of-the-practice physics and rendering engine.
5) Multiplatform distribution.
6) Low cost.

Unlike traditional engines such as Quake Engines and Unreal Engines, Unity did not come with a full game while only a simple demo was included in the engine kit. This implied that the engine is less restricted to one or several genres, and it actually supports the development of various genres from 2D to 3D without the needs of major
modifications to the engine. Another important advantage of Unity is the platform compatibility. Unity supports a long list of target platforms, ranging from desktop OS to handheld devices and even web browsers. Moreover, apart from the high usability, the world editor of unity is very easy to customize and extend. Adding new features to the editor in most cases only needs some script programming, and no modification to the source code is needed, nor is the compilation. These advantages have made Unity engine the most popular game engine for indie-developers.

### 3.3.3 Torque Engines

Torque engines were developed and supported by the American company GarageGames. The engine serials so far include Torque 3D, Torque 2D and iTorque 2D.

Torque 3D is a full-scale 3D engine with state-of-the-practice rendering and physics engines. The editor comes with the engine is a tool suite that integrates the world editor, the shader editor, the terrain editor, the physics editor and so on. Torque 3D supports a proprietary scripting language named torque script, whose grammar is similar to C++, but is dynamic-typed meaning that the type information is available for read and write at run-time. Torque 3D is not as popular as Unreal or Unity engines in the commercial development, although its cost is comparably low. But a significant advantage of Torque 3D is that since September 2012, the engine has been released under MIT license, meaning that its source code is fully accessible without additional cost. Noting that the cost to access the source code of Unreal engine is $19/Month+ 5% royalty of gross income, and the access to the source code of Unity Engine needs a special license through individual negotiation.

Torque 2D, as its name reveals, is the 2D version of the Torque engine, which shares the torque script engine with Torque 3D, while has its own rendering and physics engines. The editor coming with Torque 2D named Torque Game Builder (TGB) is also optimized for 2D game development, which integrates the world editor, the 2D animation editor, behavior editor, physics editor, etc. Unlike 3D engines, full-scale commercial 2D engines with complete SDKs are rare in the market, so Torque 2D has its unique advantage in 2D game projects. It is especially valuable for educational and academic purposes, because 2D assets are much easier to create than 3D, thus the development efforts can be saved. iTorque 2D is the variation of the engine for iOS development, and it is quite similar to Torque 2D despite of the different target platforms.

### 3.4 Engine-Based Game Development

We have gone through what the game engine is, and some typical engines. In this section, we will have a look at how game engines are applied in real game development from the technical perspectives.

The architecture of engine-based game development is illustrated in Figure 3.2.
The run-time engine at the bottom of Figure 3.2 provides the common technologies such as real-time rendering, physics simulation, and animation, to visualize the game content, and to support the players-game interaction. The game content is represented with the assets in the middle of Figure 3.2. The assets are game-specific, which usually take the most of the developing effort to create. There are mainly three kinds of assets to be created, let us look at a scenario to understand the assets.

Imaging in a FPS game you encounter a hostile Non-Playable Character (NPC) at a specific place, and he or she shoots a bullet to you. The assets used here are:

1) **Art assets**: In the scenario, they include the skeletons, the meshes and the textures of the characters, the weapons and the bullet. They also include the background music, the shooting visual and sound effect, etc. So art assets are the artistic artifacts in digital forms that feed the run-time engine for presenting the game world.

2) **World data**: The world data logically represents the scenario with the information about a) the characters and the weapons and b) the encountering environment including the sky on top of the characters, the terrain that may allow you to cover yourself, the background music that renders the intense atmosphere, and so on. So the world data is the logical representation of the environment and the game objects.

3) **Gameplay code**: The behavior of the NPC, e.g. shooting in the scenario is specified by the gameplay code. The gameplay code implements the logic behind the game world that includes the control scheme, NPC behavior, interaction flow, and etc. The gameplay code is what makes the game world live and interactive.

It worth noting that these game assets are not isolated, instead they are related to each other: the game world data refers to the art assets to specify the visual and acoustic attributes of the game objects and the environment; the gameplay code manipulate the
world data to enable the interaction and simulate the game behaviors; and the art assets are also accessible to the gameplay code, so that they can be used to change the look and feel of the game elements dynamically.

The blocks above the game assets shown in Figure 3.2 are the tools for creating the game assets. Basically three kinds of tools are needed, which are Digital Content Creation (DCC) tools [9], world editor, and script editor. The art asset is created with DCC tools, for example 3D Max, and Photoshop. The game engines usually need to provide plug-ins for such tools to transform the asset created into the engine-supported format. The gameplay code requires a script editor to efficiently create and maintain it. The editor is provided by the game engine in some cases (e.g. Torsin for Torque Engines), or adapted from third party tools in other cases (e.g. MonoDevelop for Unity Engine). The script editor is presumably a stable tool, so we do not expect the modifications to it during the development. The world data has to be edited by the engine-specific tool – the world editor. Since the game world is usually divided into scenes, i.e. levels in the game terminology, the world editor is called level editor by most of the developers. As we have discussed before, the world editor is usually not only the editor of world data, but also the entrance to the engine tool suite. The modern world editor is typically built on top of the run-time engine, so its implementation may include both the engine-native code and the script code.

In indie or modding projects, the run-time engine and the world editor are usually regarded as black box so no modifications are needed to them. However, in many commercial projects, customization of the run-time engine as well as the world editor is preferred in practice, although some engine vendors such as Unity advocated the development without touching engine source. To modify the run-time engine, the IDE for the engine-native language, most likely a GPL, is needed, which is placed at the right side of Figure 3.2. Since the world editor can consist of both the script code and engine-native code, both the GPL IDE and the script editor are possibly needed to modify it. The DCC plug-ins is probably written in a GPL, so it needs the same or another GPL IDE to maintain as what the run-time engine and the world editor need.

I have now introduced the major tools and artifacts that are essential for the engine-based development, and in the next section I will add the MDGD elements to the picture.

### 3.5 ECGM Framework

Let us take another look at Figure 3.2 and ask the question: what can MDGD do in the picture? The existing MDGD approaches reviewed in Chapter 2 answered the question by replacing the script editor partly or entirely with the model editor and the code generator, with which the game models are created and (some of) the gameplay code is generated in terms of the models. But as we have discussed in Chapter 2, this approach overlooked the world data and the world editor, so how the model and the generated code are related to the world data is not resolved, leaving a gap between engine-based development and MDGD. Some approaches chose to implement the world editor as part of the MDGD tools, which can solve the problem. However, I believe it is not sensible because of the reasons described in Section 2.6.2. For other approaches, the MDGD tools and the engine tools are not aware of each other, so the link between the model
and the world data has to be built on the generated code and world data level, as the game designers has done in the traditional engine-based development. This raises at least two issues:

1) The protocol implied in the generated code including the structure and names is dependent on the code generation software, which is maintained by the programmer instead of the game designer. If people have to know the details of the generated code to associate the model and the world data, the game designer still cannot solely finish the task, which has voided one important advantage of MDGD.

2) The code generation rule is a variable factor, meaning that the code structure and names can be changed when the code generation software is updated. This will destroy the existing associations between the world data and the model, and rebuilding such associations can be difficult, and even disastrous in extreme cases.

The above discussion is about the results of the tool interoperability problem in Section 2.6.2. To solve the problem, MDGD solutions must make the MDGD tools and engine tools interoperable to support the direct link between the model and the world data, without going through the generated code to achieve two goals: 1) the generated script code becomes transparent to the modeler and the game designer, eliminating the communication overhead, and 2) changes to the code generation software do not destroy the existing associations between the model and world data.

ECGM is a reference solution to the problem, and Figure 3.3 illustrated the architecture of ECGM.
Figure 3.3 ECGM Architecture

The box on top of Figure 3.3 represents the MDGD meta-tools, which can be an integrated language workbench or a collection of separated tools. The DSL developers use the meta-tool to create the meta-model, which is the core of the game DSL. Based on the meta-model, two tools can be created with the meta-tool: the model editor, and the code generation/model transformation tool. With the model editor, gameplay developers, e.g. the game designers, can create the model specifying the game-specific elements using the game DSL.

So far the approach has no difference from the exiting MDGD approaches. What characterizes ECGM is that the game model will not only be transformed into the script code, but also be transformed into the world data. The association between the game model and the world data is then built up within the process. Once the game model is changed, corresponding changes will be made automatically in the world data by the code generation/model transformation tool, and the game designers do not have to be aware of the script code generated, but can solely focus on the game model and world data. The process is illustrated with an example below.

Assuming a scenario in a MDGD project, where the level transition logic is modeled with a DSL, and the level layout is specified in the world editor. We elaborate the example in Figure 2.5 with Figure 3.4, which is about the level transition of World 1-2 in Super Mario Bros.

Figure 3.4 Level Transition DSL Model and Level Design View Of World 1-2 in Super Mario Bros
Figure 3.4(c) shows part of the scene of World 1-2 in a supposed world editor, say “SWE”. The scene showed three pipes, through which the player can exit World 1-2 and move on to World 2-1, 3-1, and 4-1 respectively. Figure 3.4(a) shows the DSL model of the level transition logic, where the double-edged circles embedded in the box labeled “World 1-2” represents three possible exits that connect to three world boxes. The DSL model is quite straightforward so I will not detail it more. The model will be transformed into the script code as shown in Figure 3.4(b), where a class (World_1_2) is generated for the scene (World 1-2). Three variables labeled p1, p2 and p3 are generated for the three exits in Figure 3.4(a) respectively. Since the DSL model does not contain any spatial information about the exits, it has to be specified in the SWE. The DSL model editor and the SWE are isolated in this example, so the SWE holds no knowledge about the model, thus the associations between the representation of pipes and their behaviors has to be mediated by the script code. To built up the associations, manual operations in the SWE has to be done, which can be:

3) Create three “placeholder” objects in the SWE. One placeholder shown in Figure 3.4(c) is a red transparent box overlapped with the pipe labeled “4”.

4) Modify the properties of the placeholder object, set its name to “p1” so that the generated script can recognize the object and execute the expected code in certain conditions.

To do the task, the game designer has to know the protocol such as the name (p1) of the placeholder object, and the type of the placeholder object, which is implied in the generated code. This means that the game designer needs the knowledge about the script code. Moreover, once the code generation rule changes, the protocol may change correspondingly, e.g. it is possible that the name has to be “p_1” after the change. The change then destroys the existing associations so extra jobs have to be done (change the properties of the placeholder in SWE). But if we choose to use ECGM, the process can be significantly simplified as what was illustrated in Figure 3.5.
In ECGM, the level transition model will not be transformed into the script code, but also be transformed into the world data. Three placeholder objects were automatically added to the world data after the model transformation as what was shown in Figure 3.5(c). The job left to the game designer is to drag and drop the placeholder objects to the right place and adjust its size if necessary. The properties of the placeholder objects has been set to reflect the relationship to the script code during code generation, so the script code is totally transparent to the game designer. Even the code generation rule changed in the future, the corresponding change to the placeholder objects will be done by re-doing the code generation, no manual modifications to the world data is needed, making the change irrelevant to the scene design.

The ECGM concept is easy to understand, but its implementation may not be that easy. The major problem is that the world data is engine specific, so the difficulties can be different for different engines. It can be very time consuming to figure out how to manipulate the data format of the world editor. Having the access to the source code of the world editor is important if the data is in a binary format. If we look ahead to the future of MDGD, the supports from game engine vendors such as making open interface in the world editor can be very helpful for implementing ECGM. In Chapter 4, I will further illustrate the ECGM framework by presenting a case study: RAIL and Orc’s Gold.

3.6 Summary
Game engines emerged with the first generation of 3D games such as Doom and Unreal. Today developing games based on game engines is the mainstream approach in the industry. From the software architecture perspective, a modern game engine mainly consists of two parts: the run-time engine, and the world editor on top of the run-time engine. The run-time engine implements the common technology required by various
games such as 3D rendering, scripting, and physics, while the world editor provides a WYSIWYG way to edit the game scenes, and also provides the entrance to some other tools in many cases. Due to the tradition of engine-based development, MDGD must be able to cooperate with the game engine in order to smoothly introduce it to the industry.

Some of the existing MDGD approaches were aware of the game engine, however, all of them regarded the run-time engine as the full game engine, while the world editor was overlooked. The world editor has no knowledge about the models created by the MDGD tools, leaving a gap between the world data and the models, which has to be resolved by manual operations in the world editor. In this chapter I have presented ECGM, a hybrid approach that bridges the MDGD and engine-based development to address the drawback. ECGM uses code generation or model transformation techniques to not only generate the gameplay code but also generate the world data that can be manipulated in the world editor thus closed the gap between the MDGD tools and the engine tools. The approach was illustrated by an example of level transition modeling and will be further explored in Chapter 4 in a case study.
ECGM is an approach bridging the MDGD and the engine-based development. In this chapter I will present a case study to validate and evaluate the approach. The case study is about a DSL named RAIL for modeling AI in adventure games, and a tool chain based on Eclipse Modeling Framework (EMF) technologies. The meta-model of the language is created with ECore language provided by EMF, and I choose the tree view as the concrete syntax. A model editor was generated using EMF, which was integrated into the Eclipse platform. Torque 2D was chosen as the game engine to cooperate with, which has a full functioning world editor named Torque Game Builder (TGB) and a state-of-the-art run-time game engine. I used Acceleo to create the code generator, and two kinds of artifacts were generated, which are Torque script code implementing the modeled behavior, and the world data that is compatible with TGB.

The DSL, tool chain and Torque Engine successfully worked together in developing a prototype named *Orcs’ Gold*. The results showed that making MDGD and game engine interoperable following ECGM framework is a possible, feasible and promising approach, which can significantly save development time, smooth the workflow and simplify the job of the game developers.

### 4.1 Problem Domain of the Reactive AI Language (RAIL)

RAIL is a DSL aiming at modeling the character behaviors, i.e. the high-level AI of characters in action/adventure games. The behavior to be modeled follows an event-reaction pattern. Figure 4.1 illustrated a typical behavior.

![Figure 4.1 In-game Screen Shots of *Mark of Ninja*](image)
The screenshots in Figure 4.1 are taken from Mark of Ninja (Klei Entertainment, 2012), an Action Stealth game about a ninja (player) breaking into the enemy bases to complete his missions. The three screen shots from the left to the right show three sequential states of the guard NPC on the left of the screen. The first state is *patrol*, where the NPC walking in a pre-set path repeatedly and waiting for any event that is relevant to him. Once such an event occurs, as the screen shot in the middle of Figure 4.1 shows, the NPC enters the *alert* state. The event triggering the state transition can be named “See Player”, which denotes the player character (the ninja on the right of the screen) entering into the sight cone of the NPC. In the *alert* state, the NPC shouts to alert the alien NPCs and calls for help, and he also tries to locate the player and decides the next move. Then the NPC will enter *attack* state if the player keeps visible for a long enough period, as shown to the most right in Figure 4.1. In the *attack* state, the NPC may run to the player, and/or shoot at the player with his gun, or use other suitable weapons. What actually happened in Figure 4.1 is that the player was shot down, and the red dotted line visualizes the trajectory of the bullets.

![Figure 4.2 Screen Shots of Pac-Man](image)

Figure 4.2 shows another example about the behavior of the Ghost (enemy actor) in Pac-Man (Namco, 1980). The screenshot to the left shows the default state of the Ghosts when they are far from the player, in which they randomly move around the map. If they receive an event that the player (shaped as a yellow pie) is near, they will enter the chase state shown in the middle screenshot, in which they try to catch the player by running to him. But if the player somehow obtains a power-up represented as a ball, he becomes strong so that he can eat the Ghost, and when the Ghost receives the event about it, it has to enter the *Flee* state as what was shown in the right screenshot. In the *Flee* state, the Ghosts try their best to run away from the player.

We can identify some major concepts in the described behaviors:
• **AI Pattern**: The whole behavior of a specific NPC follows a pre-set pattern about what it is going to do when a given event is received in each particular condition, and what it is going to do by default when no event occurs.

• **AI Pattern State**: The reactions to a given event can be various in different conditions even for the same NPC. The “AI Pattern State” concept is the abstract of the condition that the NPC is in at a particular moment. Typical examples of it include the “patrol” state in Figure 4.1 and the “flee” state in Figure 4.2.

• **NPC Action**: The whole NPC behavior consists of many sequences of moves, of which each completes a basic task. I call such a sequence of move “NPC Action”. Typical actions can be “move to a location”, “shot a bullet at the player”, “run to the player”, etc. Actions usually are the responses of the NPCs to events.

• **Event**: The action of NPCs is stimulated by an event or an event composite. An event can be directly connected to the player behavior, e.g. “the player enters vision”, “the player becomes invisible”, and “the player is aiming at me”, or it can be not directly connected to the player, e.g. “the boss is down” and “the light is off”. An event can trigger an action, and/or other events, and the event-action chain defines complex behaviors.

RAIL is designed to support the modeling of the reactive AI in terms of the above concepts. Note that RAIL is intended to use the concepts as the building blocks for modeling, but not to model the concepts themselves. E.g. “Walk to a place” may be a building block of RAIL models, but how a NPC walk to a location may involve path-finding and obstacle-avoiding algorithms and animation playback control, which is not going to be modeled with RAIL. Therefore the problem domain of RAIL is not an entire game, and even not the whole gameplay system of a game, however it is still valuable in the practical development, because:

1) The high-level behavior is game-specific, and can take much development resource to implement with the traditional coding technique. So far there is no common implementation of such behaviors, so they have to be implemented individually for each game project. Regarding the development workload, we look at an example about the development of SplinterCell:Double Agent (Ubisoft, 2006) that I participated, where about half of the programmers including myself were working on the high-level behavior throughout the project.

2) The concepts are not fully supported by the existing GPLs and scripting languages. There do exist script languages that provides partial support for the concepts, for example, Unreal script supports the “state” concept at the language level. However, a DSL like RAIL can further raise the abstraction level and make the solution even simpler.

All in all, the target domain of RAIL can be summarized as: *The high-level NPC behavior based on the domain concepts such as pattern, state, event and action for Action/Adventure games.*
4.2 Language Definition of RAIL

The abstract syntax and static semantics of RAIL are defined with an Ecore meta-model as shown in Figure 4.3. Some low-level details were omitted to make the meta-model better focused and more readable.

![Figure 4.3 RAIL Meta-Model (Simplified)]

The language adapts some core concepts from State Machine to the AI modeling domain. The top-level RAIL construct is Game, which is the container of all AIPatterns in a game. Each computer game to be modeled should have one and only one instance of Game. AIPattern is the central construct of RAIL models that corresponds to the AI Pattern concept in the domain description in Section 4.1. Modelling with RAIL is mainly about creating various AIPattern instances, each of which defines a particular kind of NPC behavior. AIPattern is stateful, meaning that the NPC reactions to events are influenced by the condition that the NPC is in at a particular moment. The AI Pattern State is the abstract of the condition in the domain description, and the State is the corresponding construct in the language. Each AIPattern possesses a group of State instances reflecting all the possible conditions that are relevant to the reactions of the NPC following the AIPattern. But an AIPattern can only be in one State at a moment,
say the “Active State”, and the initial Active State is represented by the EReference named “default”. A special case of AIPattern is that it has only one state, then the state can be omitted and the Triggers (described later) will be directly connected to the AIPattern.

A State has a group of triggers, which defines the actions to be performed in reaction to a stimulus that is typically an event or a composite of events. For example, a trigger can be:

\[
\text{Event(“See Player”) -> Action(“Move to The Player”)}
\]

Example 1

or be more complex:

\[
\text{Event(“See Player”) OR Event(“Hear Player”) -> Action(“Alert Aliens”) AND Action(“Move to The Player”)}
\]

Example 2

The Event construct can be further elaborated with vision events, input events, AI interactive events, etc. I will not go into the details about these events here. The stimulus can also be something other than the Events, for example, state change, pattern initialization, and logic operations.

The Action construct encapsulates the actual actions to be performed by the AI pattern as the result of the stimulus. A common kind of actions is IssueCommand about sending a specific command to the NPC controlled by the AI pattern, such as ”Move to A Location”, ”Attack A Target”, and ”Look at A Place”. The command will be performed by script code related to the NPC, and the script code is part of the domain framework being developed with manual coding. The Pattern Control action is intended to provide a way to manipulate the AIPattern at run-time. A typical example is ”Go to State” that sets the current State of the AI pattern to the one specified in the Action parameters. Other pattern control actions include ”play a sound effect”, and ”update gameplay status” etc.

The instances of the Trigger not only can be properties of States, but also can be directly associated with AIPatterns, and then they become ”Default Triggers”. The Default Triggers will take effect in any State, and if they are in conflict with the State-owned triggers, they have the priority.

---

1 EReference is the language construct in ECore language representing the relationship between the elements in a ECore model.
The concrete syntax of RAIL is based on a tree-view. I choose this form because the AIPattern-State-Trigger Action/Event hierarchical relationship naturally follows a tree structure. Figure 4.4 shows a RAIL model within the Eclipse-based model editor.

![RAIL Model in Editor](image)

**Figure 4.4 A RAIL Model in Editor**

RAIL is a DSL similar to a state machine, but has its unique characteristics. State machine as a design pattern [68] is widely used in gameplay programming, and Unreal Engine has provided the language level support for it. RAIL borrowed the core concepts from the state machine, and made some important modifications to adapt them to the game modeling domain, such as:

1) Defined domain-specific constructs to describe game-related concepts.
2) Provided Trigger concept, which can represent the reaction in a nature way.
3) Supported global reaction rules that will take effect in any state.
4) Supported a simplified AIPattern with only global triggers.

Note that RAIL is not intended to be a unified solution for modeling the gameplay of all the action/adventure games, instead, it should be customized for each particular game families or even a single game project. But the high level structure of the language, i.e. the constructs showed in Figure 4.3 should be able to be reused without major modifications.
4.3 Tool Chain Implementation

To enable RAIL-based gameplay development, I implemented a tool chain based on the Eclipse/EMF platform. Two tools were implemented: The model editor and the code generator. EMF has provided a framework, with which the model editor can be automatically generated from the meta-model. The code of the generated model editor is in java, and is deployed as an Eclipse plug-in, so that the tool is integrated with the Eclipse IDE. The java code can be modified for any customization purpose, for example, changing the appearance of the language constructs, and optimizing the user interface. The default concrete syntax that the generated model editor provides is the Tree View, which is exactly what RAIL needs, so extra work in customizing the generated editor was not needed. Figure 4.4 shows a screenshot of the RAIL model editor.

The code generator, on the other hand, required much more work to create. There exist various frameworks on Eclipse platform for code generation, such as Xtend and ATL. I chose Acceleo (https://eclipse.org/acceleo/) to do the job. Acceleo has been developed by the Eclipse Strategic Member Obeo since 2005, which is included in the Eclipse release train since Eclipse 3.6 Helios. Acceleo provides a template language for creating code generators following the template and meta-model approach [60], which supports any Ecore-based meta-model (other meta-models as well) and theoretically can generate code in any target programming language. Acceleo also provides extensive language tools helping the code generator development, such as a template code editor, a debugger, and a profiler. Many advanced features are supported by Acceleo, for example, syntax highlighting, code folding, and refactoring. I chose Acceleo mainly because the template language is easy to learn and use, and the documentation is sufficient. The code generator of RAIL was then implemented as a couple of templates, which took RAIL models as input and generated code in Torque script for Torque 2D game engine that I will discuss in Section 5.4. The code generation templates can be created and executed in the Eclipse platform since Acceleo is an Eclipse plug-in.

4.4 Integrate RAIL with Torque 2D Engine

Torque 2D is a commercial game engine developed by GarageGames (www.garagegames.com), which supports developing various genres of 2D games from platform games to RPGs. The game code for Torque 2D is mainly written in a script language called “Torque Script”, which has a C-style syntax plus some object-based features. Torque 2D engine provides a powerful world editor: Torque Game Builder (TGB). TGB organizes the game world with scenes (levels), and the scenes can be created in a WYSIWYG way.

There are mainly two tasks to be done to integrate RAIL with Torque 2D: 1) raise the abstraction level of Torque 2D, and 2) implement the generator for script code and world data. I will detail them in the rest of this section.

4.4.1 Raise the Abstraction Level of Torque 2D

Since RAIL was designed only for modeling high-level AI of Action/Adventure games, it targets a narrower domain and lies on a higher abstraction level than Torque 2D.
Therefore an abstraction layer must be implemented on top of the Torque Script to promote Torque 2D to a suitable domain framework [8].

The abstraction layer was implemented as a Torque Script library, where several concepts were implemented using an Object-Oriented approach. Character is the core module of the abstraction layer. Character simulates the creature or the machinery in a game that can perceive surroundings and react to the environment. Character is both an event source and an event handler: Character detects other objects every frame, using the perception simulation algorithm, and generates corresponding perception events. The events were typically sent to the AI layer (modeled with RAIL) as the input for decision-making. On the other hand, Character is also responsible for handling the events from its Controller that are typically commands like move and attack. The Controller can either be the Player Controller that translates player inputs into commands, or be the AI Controller that is modeled with RAIL.

Character mainly implemented four kinds of algorithms:

1) *Perception simulation*. It is the algorithm used to simulate vision and hearing of creatures or non-creatures. Character uses the perception simulation algorithm to detect the objects in the perceivable range and gather their perceivable information.

2) *Path finding*. It is the algorithm that guides a Character to travel from one spot to another in the game scene. It is essential for Character movement.

3) *Movement simulation*. It is the algorithm that calculates Character position, orientation, velocity and other physical attributes.

4) *Animation management*. It is the algorithm that keeps the animation of Character consistent with its physical state.

Less important modules of the abstraction layer including *input handling, event management, and global rules*, will not be discussed in detail.

### 4.4.2 Generate Code and World Editor Data for Torque 2D

To integrate RAIL tools with Torque 2D engine following the ECGM approach, I created two Acceleo projects for generating two kinds of artifacts from RAIL models: 1) The Torque Script code implementing the modeled behavior, and 2) the data for TGB (world editor).

#### 4.4.2.1 Generator AI Code

With Acceleo, we can create one or more templates for each EClass object in the meta-model. When a template is processed on a DSL model, code will be generated according to the template for each EObject that is an instance of the EClass that the template is related to. Processing of a group of templates can form a Chain, with which you can easily generate code for the entire model.
Each RAIL model includes multiple AIPatterns, each of which defines a specific type of NPC such as neutral NPC, enemy soldier, and boss. A Torque Script class was generated for each pattern, and a couple of member functions were generated for the states and triggers possessed by the pattern.

4.4.2.2 Generate World Editor data

The Torque Script code must be associated with the graphical objects in TGB. This is done manually with the traditional coding method and existing MDGD approaches, which is an error-prone task. With ECGM approach, the code-object relationships were built automatically through a specific generator. The generator is also developed with Acceleo as templates, and the format of the generated data complies with TGB extension protocol.

TGB uses an object palette to manage object prototypes. For each object prototype, e.g. a picture, or a sprite animation, there is a visual object in the palette. Users can pick a visual object in the palette, and then create an object of the same type and initial attributes. The TGB extension protocol allows adding customized object prototypes to the palette of the world editor. We create one pattern object prototype in the TGB palette for each AIPattern in the RAIL model. Therefore the AIPattern is visualized in the TGB as a graphical object like other built-in object prototypes.

4.4.3 Tool Chain Architecture

The tools described above consist a complete MDGD tool chain for developing 2D adventure games. The tool chain uses EMF based model editor to create and modify RAIL models, and TGB to create and modify world data. The two kinds of tools were bridged with Acceleo-based generators that not only generate AI code but also generate world data. When creating game scenes, designers access the AIPattern from RAIL models through the graphical objects in TGB palette without knowing the existence of the generated code. When adding or modifying AIPatterns, developers work on models instead of code. Figure 5.5 shows the tool chain architecture.
The dotted line between Pattern A in the object palette and Pattern A in C Script Code implies the association automatically built by the tool chain. If a user wants to connect Pattern A that is modeled with RAIL to character A in a level, he or she can drag Pattern A from the world editor palette to somewhere near the character in the level. A pattern will automatically linked to the nearest character, and the association is built by the generated code as well as the domain framework. Figure 4.6 shows an example of using AIPattern in TGB.
4.5 Orc’s Gold: A Prototype Game

I created a game prototype to validate the tool chain described in Section 5.4. The game is a 2D action/adventure game named Orc’s Gold. The general game concept is that a player controls a human character to steal gold chests from orc’s camps. The chests are guarded by orc guards and dragons. The orc guard stands still at a place by default, and when a player enters his sight, the orc guard will chase the player and try to kill him. If the player is caught, he will die and the game will be over. The dragon on the other hand does not move, and it sleeps and wakes up periodically. If a player passes a dragon when it is not sleeping, the player will be eaten and the game will be over. The dragon will also eat an orc guard if it goes through the dragon-guarded place carelessly. So players can trap the orc guards by leading them into the mouth of dragon, and take their chests afterwards. The player can walk or run. When he is running, he moves faster than the orc guards, so he can also attract the attention of the orc guards and then take the chests by wisely taking the advantage of speed. If a player successfully steals all the gold chests on the map, he wins the game. Figure 4.7 shows a screenshot of Orc’s Gold.
The game uses four AIPatterns to control the characters, which are Orc Guard, Dragon, Chest and Tree. Apart from the first three patterns that have been described, the Tree pattern as its name revealing, controls the trees on the grassland. The trees are purely decorative objects for rendering the gaming atmosphere that almost have nothing to do with the gameplay, but they will play a “swing” animation when a player touches them.

To evaluate the productivity impact of RAIL modeling, I implemented the four patterns with two methods: the manual coding method and the MDGD method. The time used and Lines of Code (LOC) developed for the two methods are summarized in table 4.1.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Used (Hours)</th>
<th>Manual LOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Domain Framework</td>
<td>AI Patterns</td>
</tr>
<tr>
<td>Manual Coding</td>
<td>20.5</td>
<td>9.5</td>
</tr>
<tr>
<td>MDGD with RAIL</td>
<td>0.7</td>
<td>0</td>
</tr>
</tbody>
</table>

In Table 4.1 we see that the domain framework including the low-level AI, input handling and other low-level mechanics took more time and LOCs to develop than the high-level AI Patterns. However since the domain framework is used in both methods, the cost is also shared by them. Moreover, since the domain framework is on a low abstraction level and less relevant to specific gameplay, it is reusable for future AI Patterns and even in future games.

The time used on the AI Pattern development is 9.5 hours with the traditional manual coding method, and it is dramatically reduced to 0.7 hours with the RAIL-based MDGD
method. Regarding the LOCs, I wrote 357 lines of C-script code to manually implement the AI Patterns, and with MDGD method, all the AI Pattern code is automatically generated from a RAIL model, and no manual coding is needed. According to the data, there is a significant productivity improvement from using RAIL-based MDGD, and the result is also inline with the reports of MDGD from other software domains such as [89-91].

The benefits of DSM are not free, and the initial investment must be paid for developing the DSL and the corresponding tools[24]. In table 4.2, I present the time and LOCs used in creating RAIL and its tool chain.

<table>
<thead>
<tr>
<th>Table 4.2 Cost of Developing RAIL and Its Tool-Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (Hours)</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>RAIL and Its Tool-Chain</td>
</tr>
</tbody>
</table>

As it was discussed in [24], the initial investment on DSM can be paid back by repetitively use of the DSL and the tools created, and by lowering the technical threshold for the developers. The more products and variants created with the DSL tools, the faster the investment is paid back. In the RAIL case, interestingly the investment is paid back in just one product: if we add the cost of developing RAIL tools to the cost of developing the prototype, the total cost is still lower than the manual coding method. This may because of two reasons: 1) the RAIL lies on a proper abstraction level that can significant improve the productivity while keeps the language simple for implementation and 2) the use of EMF and Acceleo framework significantly improved the productivity of developing the DSL tools.

Thanks to the ECGM approach, RAIL-based development enabled an efficient workflow. The level data and artistic data was created and managed using the Torque world editor, and the high-level AI code was generated from the RAIL model that was created using EMF-based RAIL tools. The engine tools and the MDGD tools can thus focus on their native domain respectively, and the different artifacts were developed in the most appropriate way. Moreover, as I have described in Chapter 3, ECGM features the creation of level data from the DSL models, which bridges the code generated from the DSL models and the game world in level editors. In the Orc’s Gold prototyping, this was implemented with generating four AI Pattern placeholders in the Torque level editor, which corresponded to the four AI Patterns in the RAIL model respectively. Connecting a generated AI Pattern code to a character in game scene was as easy as dragging and dropping a placeholder object from the object palette to the scene, and putting it near the target character. Level designers only have to know which placeholder object corresponds to the desired AI pattern, while the generated code is simply transparent to them. Changing the rule of code generation also has no side-effect on the level design.
By analyzing the logic of the generated code and the manually written code, the performance of them is expected to be equivalent, because the algorithms and the mechanics implemented with two methods are actually identical. The results of profiling also support the impression, where I played through the two versions (generated code and manual code) of the game in the same way on the same computer, which took 88.29 second for the manual version and 82.72 second for the generated version. The average Frames Per-Second (FPS) measured in two gaming process were exactly the same: 31.24 FPS.

The RAIL-based modeling method also improves other aspects of software quality, such as modifiability. For instance, since the C-script does not fully support object-oriented programming, the manual code sometimes has duplicated parts spreading among several modules, and when the duplicated code needs to be changed, the same modifications have to be done several times on different modules. This is an error-prone task. Thanks to the language features provided by Acceleo, the problem can be solved at the code generation level in RAIL-based modeling: the duplicated parts of the generated code can be generated from one code-generator module, and modifying the module will result in the modifications on all the generated modules with the duplicated parts. Generally, the language for writing code generator provides an extra means to compensate the drawbacks of the target language for well-modulizing the generated code.

### 4.6 Discussion and Related Work

The results of the case study showed that RAIL and its tools can significantly reduce the time and code lines needed in the development of Orc’s Gold. The cooperation of engine tools and MDGD tools enabled an efficient workflow, which validated the usefulness of ECGM approach.

The initial investment of model-driven development is a general concern. To minimize the problem, ECGM embraces engine tools, and narrows the scope of modeling to an appropriate level, thus reduces the requirements to the modeling language and tools. The use of language workbench, i.e. EMF and Acceleo also significantly reduced the initial investment in the practical aspects. The case study showed that the initial investment on the meta-model and code generator for RAIL was acceptable, and it was paid back in just one project. Moreover the tools can be used for creating many more patterns for extending Orcs’Gold to a real game, and even be able to be reused in other 2D action/adventure games.

Splitting the low-level AI and the high-level AI is necessary in game modeling. Script languages or GPLs are appropriate for implementing low-level AI, because they were just created for solving the problems on this abstraction level. For example, the major problems with vision simulation is 1) find objects in a vision cone, and 2) check the collision between the line of sight of a NPC and the other objects in the game world, which are both geometric problems and are appropriate for GPLs. However, DSL is appropriate for creating the high-level AI, because the problems involve a lot of
domain-specific concepts, which is not possible to be directly supported by the low-
level languages. The case study showed that low-level AI costs a lot of development
effort, and this may raise a question that if ECGM has solved the most difficult
problems. But it worth noticing that since the character layer is reusable among patterns,
even reusable among games, so the cost does not scale with the project complexity.
However, the cost ECGM has reduced is the scalable cost, which is more important in a
real game project.

I also want to point it out that when integrating models with engine tools, source code
level modifications may be needed to the engine tools. Some engines are open to
extensions, which can make the integration trivial. But although Torque 2D is intended
to be open, there are still some issues needs to be fixed at the source code level. The
modifications I made were minor (3 LOC), but it brings a problem that every time we
update the engine, we have to redo the modifications. So for the engine vendors, it is
wise to keep the format of level data flexible and open to external tools, because this
can make the engines have better opportunity to be chosen as the target engine in
MDGD projects.

In Chapter 2 I have presented the MDGD approaches in literature, and pointed out a
major drawback that is the engine-integration problem. Among those MDGD
approaches, Pleuß and Hußmann's approach[21, 45, 46] shares the same underlying idea
with ECGM. They integrated MDGD with authoring tools, more specifically, Adobe
Flash. Similar to our approach, two kinds of artifacts were generated: script code
(ActionScript) and media objects (FLA files), and they were directly associated. Their
work focused on the DSL, and they did not conceptually define a MDGD approach as I
did. Furthermore, if we compare their work with mine, a major difference here is that
ECGM focuses on the integration of MDGD with commercial game engines instead of
multimedia authoring tools, which further reduced the gap between the general MDD
methodology and the specific game development domain. Moreover, the differences
between the modeling languages, and the differences between the modeled domains are
also the aspects make our contributions unique.

4.7 Summary
Model-Driven Development is a trend within software engineering, and I believe it will
also help game industry in the future, although significant issues still remain to be
resolved with the existing approaches. ECGM approach pushes the development
method one step forward to the industrialization of MDGD.

In the case study, ECGM is evaluated through implementing RAIL and its tool chain,
and prototyping Orc’s Gold, a 2D adventure game. RAIL is a DSL for modeling the
high-level AI in action/adventure games, and its tools including a model editor and two
kinds of code generators that were implemented with Eclipse/EMF technology and
Acceleo. The results confirmed the feasibility and effectiveness of ECGM. The RAIL
will be extended to support diversify game architectures later in this thesis, and further
work also includes implementing a tool chain for other engines to validate ECGM, and
refining the RAIL to increase the usability, which are however not covered in the thesis.
Software architectures of video games have evolved radically over the last decade. Researchers and practitioners have proposed architectural patterns such as stand-alone configuration, Peer-to-Peer, Client Server and Hybrid. As I have discussed in Chapter 2, all of the existing MDGD approaches only have addressed stand-alone architectures while an important fact has been overlooked that most of the today’s computer games in market more or less have networked features that have to be supported by more complex architectures. The lack of complex architectural support leaves a gap between the MDGD approaches and the commercial game development.

To close the gap, the DSLs used for MDGD must integrate the concepts from the software architecture domain, and a conceptual framework is thus needed to support the design and development of DSLs. In this chapter a conceptual framework for game architectures – Game World Graph (GWG) – is presented for this purpose. The corner stones of the framework are the Game World and the World Connector concepts, which reveal some essential characteristics of game architectures. The framework is validated by a literature review including 40 game architectures before applying in the MDGD design, which is documented in Chapter 6. Apart from the conceptual support for MDGD, the framework has at least three secondary functions: 1) classifying existing architectures, 2) communicating and sense-making of game architectures, and 3) exploring and discovering future game architectures.

5.1 Game Architecture and MDGD

Since the first popular commercial game Pong was released in 1972 [92], game developers have created video games with an increasing amount of content. Today, game software has a much higher complexity compared to games developed 30 years ago. The evolution of game software architectures from 1994 to 2004 was illustrated in [1], where a typical 2D game from 1994 consisted of five main modules compared to over 30 modules for typical massively multiplayer 3D game in 2004, where each module consisted of between six thousand to forty thousand lines of code. In line with the increasing game software size and complexity, the architecture patterns used by game developers have also evolved over time. Before the early 1990s, most games only consisted of a single application, where all computations were performed on the
player’s machine. This stand-alone architecture has dominated the game market for over forty years and is still the dominating architecture for many successful commercial titles today. In the middle of 1990s, networked games like Command and Conquer (C&C) and Warcraft became popular. The implementation of these games was based on the Client-Server architecture pattern, and the software was running on multiple machines connected through LAN or the Internet. The retailed game software integrated both the client and the server part of the game, so each player could either host a game on his or her machine, or join a game as a guest. The exponential growth and development of the Internet at the end of last century enabled the development of Massively Multiplayer Online Games (MMOG). Most MMOG software uses a powerful centralized server or a server farm to simulate a complex and sharable game world (virtual world), allowing multiple players to connect to the game world through the Internet with their client machines (e.g. an ordinary PC). The distribution of computation and the use of professional servers provide more computational power, which significantly improve the richness of the offered game worlds. Moreover, the staggering number of online players greatly extends the social aspects of video games and creates new opportunities for social gameplay.

Challenges have emerged during the evolution of game architectures. Work of researchers and practitioners has pointed out that poor robustness [93] and lack of flexibility [94] are among the most important challenges of the Client-Server architecture pattern. On the other hand, the Peer-to-Peer (P2P) solution, which was mainly proposed by researchers as an alternative to Client-Server, also had its drawbacks such as the difficulty in preventing cheating [93] and maintaining consistent game states on peers [95]. These challenges have motivated for further research in the game architecture field, and some sophisticated solutions have been proposed addressing these issues.

Since the game architectures vary, MDGD must not overlook the diversity of the game architecture, otherwise it can hardly be accepted by the industry. However, a significant drawback of the existing approaches is that all of them only concerned with the stand-alone architecture. This is thus a major motivation of my Ph.D. research. As I have discussed in Section 2.7, game architecture influences at least three aspects of MDGD that are 1) DSLs, 2) Code Generator and 3) Domain Framework. Therefore existing MDGD approaches has to be revised in these aspects to address the architectural diversity. To address this problem, work on both conceptual level and technical level has been done in my research, and the former is presented in this chapter and Chapter 6, while the latter is presented in Chapter 7.

The goal of the conceptual level work is to investigate the game architecture domain and make a conceptual framework that can help design DSLs supporting the modern game architectures, thus close or reduce the gap of game architecture support between the existing MDGD approaches and the commercial game development. To create such a framework, I started from looking into the representative game architectures and then tried to classify them. The taxonomy used in the classification implies a conceptual framework, because the criteria used in the classification must be able to capture the characteristics of different architectures, therefore concepts on game architecture
domain can be identified from them, which constitute the base of a conceptual framework.

I did not use the architectural patterns such as client-server, peer-to-peer from the general software architecture domain as the base for the taxonomy, because game architectures are domain-specific, whose differences sometimes cannot be captured by the terms from the general software architecture domain. E.g., there are several different proposed architectures that share the same name “hybrid architecture” [94-96]. The general client-server architecture also has many variations in game software domain such as [97], [98], [99], which can hardly be classified as one kind. Therefore I defined Game World and Connector concepts and used them as the cornerstones, basing on which a conceptual framework, Game World Graph (GWG), was built. GWG as a conceptual framework has many applications including at least following:

1) It provides a terminology for communication on game architectures, which is more specific and appropriate than the terms borrowed from the general software architecture domain.
2) It can be used as a taxonomy framework for classifying and analyzing game architectures. In Chapter 6, I will present a literature review including 40 game architectures to demonstrate GWG as a taxonomy framework.
3) It can be used to explore new game architectures. This will be demonstrated in Section 5.3.4 with an example.
4) It can be used in MDGD to provide conceptual support for DSLs and code generators. In Chapter 8 I will demonstrate how to integrate GWG into MDGD with a case study.

In this chapter I mainly discuss the first three applications, and Chapter 8 focuses on the application of GWG in MDGD. The rest of this chapter is organized as follows: Section 5.2 introduces the GWG framework, Section 5.3 describes how the GWG framework can be used as a tool to classify existing architectures and to explore future architectures, Section 5.4 discusses the quality of GWG framework, the GWG Composite, and compares the framework with other taxonomies, Section 5.5 concludes the chapter.

5.2 The Game Worlds Graph Framework

The initial idea of the GWG framework came from a preliminary research of exiting game architectures in order to analyze their characteristics, including their advantages and drawbacks. When I was looking into some modern game architectures, I found it difficult to distinguish between them using traditional taxonomies, e.g. typical Client-Server architecture versus Remote Rendering architecture. By examining these architectures I found that the game state distribution and connections between the game states are their essential differences, which inspired me to create a new architectural viewpoint: The Game World Graph.

In this section, the Game World and the World Connector concepts serving the cornerstones of the GWG framework are defined. Further, the GWG formalism based on graph theory is described. Finally, the Graph Style and Flavor are defined as
abstractions of commonalities among GWGs, which are further used as the basis in the conceptual framework.

5.2.1 Game World

We use term “World” as a metaphor to describe the game state of a video game, which is created and updated by the game program to simulate an interactive environment and its interactive objects. The Game World logically denotes all the dynamic objects (player characters, non-playable characters (NPCs), weapons, etc.) and static objects (buildings, rooms, sky, and so forth) in a game that are computationally represented in system memory, and updated at run-time by the game program to reflect the newest simulation result. The Game World concept is not my invention, and has been widely used in previous research articles in terms like “World” [100], “Scenario of Game” [100], “Game State” [101, 102], “Game Gaming World” [103], “Game World” [104], and “Virtual World” [105].

Single player games and multiplayer games played on one console/PC normally have only one Game World. However, multiple Game Worlds can be identified in a game instance when looking at games with more complex architectures. E.g. a multiplayer game based on a Client-Server architecture pattern may simulate a shared remote Game World on the server, and multiple local Game Worlds on the players’ clients. Synchronization protocols are required to keep the client Game Worlds updated and consistent.

In the Client-Server example above, the Game World on the server is different from those on the clients at least for two reasons: 1) The Game World on the server contains the complete state of the game, while the Game Worlds on the clients may only contain a local state related to a specific player; 2) When the Game Worlds are synchronized, the Game World on the server normally has higher authority that dictates the content of the Game Worlds on the clients. Neither the Game World on the server, nor the Game Worlds on the clients can be perceived directly by players. Furthermore, what players can see and hear may be different from those above-mentioned two types of Game Worlds. E.g. there may be tens of thousands of player characters simulated in the Game World on a MMOG server, only ten player characters contained in the local Game World on a specific client, and a specific player can only perceive (see and hear) five of them due to distance, view direction, and rules of the game. This third kind of Game World contains objects that are perceptible to a specific player, reflecting only externalized attributes of these game objects, in a presentable format (video stream, graphics, etc.). This third kind of Game World is named as Perceptible World.

To summarize, the GWG framework differentiates between the three kinds of Game Worlds in a game with the following concepts:

- **Global World**: The Global World (GW) contains the shared global state (all player characters, NPCs, weapons, etc.) of a game instance. The GW of a game reflects the definite, instant, and complete result of the game computation. The other Game Worlds have to be updated according to the GW sooner or later to avoid inconsistencies. There is only one logical GW for one running game instance. However, when it comes to the technical implementation, the GW can
be implemented as multiple sub-GWs. Each sub-GW may contain partial state of the logical GW. E.g. in some MMOGs, the whole game world is divided into zones managed by different servers and each zone can be regarded as a sub-GW. Each sub-GW can also be a complete duplicate of the logical GW. E.g. in the Mirrored Server Architecture, the same game world is simulated on multiple servers and each server manages complete and mirrored state. The duplication or partitioning of the GW is normally due to technical concerns (reduce load, decrease network latency, etc.). Since the sub-GWs contain most attributes of the logical GW, we still use the term Global World to denote a collection of sub-GWs.

- **Local World**: The Local World (LW) can be regarded as the “cached” game state for one player. The LW contains only the game state that the game program considers “relevant” to a specific player, i.e. the virtual entities and the environment variables within a specific player’s area of interest, which the player will potentially interact with between two update events. LW reflects a dirty game state, which could be outdated or up-to-date, correct or incorrect, complete or incomplete, depending on the client computation and the synchronization protocols. The LW is usually used to keep games interactive, to save bandwidth, and to prevent cheating.

- **Perceptible World**: The Perceptible World (PW) is the game world on the player’s screen and/or other presentation devices. The PW can only contain a very limited subset of the GW or the LW due to rules of games and performance concerns. Players perceive the game only through the PW externalized by various presentation devices, thus it becomes the bridge connecting the player to the logical game world.

This is not an exhaustive classification. These three concepts may not cover all possible Game Worlds, e.g. persistent Game Worlds on external memory. We choose these Game Worlds to build our conceptual base, because to our best knowledge, the essential data and processes they capture are the most relevant to the architecture inventions and innovations in the game domain.

### 5.2.2 Synchronization between Game Worlds

The Game Worlds are not isolated and the content of a Game World is usually connected with another Game World. E.g. a LW on a client must keep its content consistent with the GW on the server. We use the term World Connector to capture the relationship between two connected Game Worlds. The term connector is taken from [86], where it is defined as “a communication vehicle among components”. A World Connector can be either native or remote, depending on whether the two connected Game Worlds are simulated on the same node (machine) or distributed on two separate nodes. The communication between game worlds is considered as update flows through the connector. Different computations as well as different types of update flows are required for particular connector types. When two Game Worlds are connected natively, they both will be simulated in memory on the same computing node. A connection normally involves transforming data from one format to another. A typical example is when a LW is connected to a PW. The graphical rendering computation (e.g. 3D transforming, rasterizing) is then required to draw objects onto the screen. On the other
hand, if two Game Worlds are connected remotely, synchronization and other network computations are necessary. For example, when a GW is connected to a LW, the program simulating LWs has to request/receive the most recent data from the GW, and correspondingly updates LWs to provide consistent local states. Note that two Game Worlds of the same kind also can be connected. For example, in a multi-server game, each server has its own copy of the GW. State synchronization computation is needed to keep them consistent. Some solutions to this problem can be found in [95]. They are relatively more complicated than the simple synchronization protocol between a GW and a LW.

To summarize, the GWG framework defines two kinds of connectors connecting the three types of worlds (global, local and perceptible):

- **Native Connector**: Denotes the computation synchronizing of two Game Worlds on the same computing node.
- **Remote Connector**: Denotes the computation synchronizing of two Game Worlds on two different computing nodes.

### 5.2.3 Game World Graph Formalism

If we consider Game Worlds and Connectors as vertices and edges, graph theory can be used to model and relate the terms. The collection of Game Worlds and the collection of World Connectors of a game can be defined as a graph. I use the term Game Worlds Graph (GWG) to denote the graph of Game Worlds and World Connectors. The Game Worlds Graph is formalized as follows:

\[
GWG = (W, C), \text{ where } W \text{ is the set of Game Worlds simulated by a game instance,}
\]

\[
\text{and } C \text{ is the set of World Connectors connecting elements in } W.
\]

**Figure 5.1 Two Examples of Game Worlds Graph**

Two examples of Game Worlds Graphs are shown in Figure 5.1. 5.1a) is a typical GWG of a single player game, where a Perceptible World (P) is natively connected to a Global World (G), meaning that the simulation and externalization of the game world is performed on a single machine. In 1a), \( W = \{G, P\} \), and \( C = \{C1\} \). 5.1b) is a typical GWG of a multiplayer online game. Three Local Worlds (L1, L2, L3) are remotely
connected to a Global World (G), meaning that each player has a local copy of the
global state on her or his client, which is usually a partial copy of the full state
maintained on a remote server. The Perceptible Worlds (P1, P2, P3) are natively
connected to the Local Worlds meaning that a client is also in charge of the
externalization of the game state for a specific player. In 1b), W= \{G, L1, L2, L3, P1,
P2, P3\}, and C= \{C1, C2, C3, C4, C5, C6\}.

Game Worlds Graph serves a new perspective to how to visualize and examine the
software architecture of a game. As it’s defined in [86], “The software architecture of a
program or computing system is the structure or structures of the system, which
comprise software elements, the externally visible properties of those elements, and the
relationship among them.”

Note that the Game Worlds Graph itself will not represent the whole software
architecture even if we can look on Game Worlds as elements of the game software, and
World Connectors as relationships among them. However, the externally visible
properties are not described. Also, “the systems can and do comprise more than one
structure and that no one structure can irrefutably claim to be the architecture”[86]. So
the Game Worlds Graph can be seen as a view of the game software architecture, but
cannot claim to be the whole architecture. Game Worlds Graph reveals some essential
architectural characteristics of a game:

1) The Game Worlds Graph provides a component-connector view [86] of the
software. The Game World encapsulates an independent set of data and
simulation processes typically represented by one or more components in the
game software. The World Connector encapsulates the synchronization and
transformation processes maintaining the connection among the components.

2) The Game Worlds Graph provides an allocation (deployment) view [86] of the
software. The types of the World Connector with which the Game Worlds are
connected imply how the software is assigned to computation nodes.

3) The Game Worlds Graph shows the runtime configuration of the game software,
which is required for analyzing parallel and distributed performance of the
game.

Note that the GWG is a runtime model of the game, and it can change from time to time.
There may not be one unique graph for a particular game or even for a particular
running game instance. E.g. consider a multiplayer client-server game. The number of
connected players is changing continuously, meaning that the configuration of the Local
as well as Perceptible Worlds is changing, thus results in a changing Game Worlds
Graph. However, the software architecture of the game does not change over time. The
GWG framework provides the abstractions Graph Styles and Flavors to provide a static
view that reflects the game architecture including a changing run-time configuration of
players, which will be described in the next section.

5.2.4 Graph Styles and Flavors
A single-player game normally has no LWs (as the GW and the LW are the same), and
this will not be changed during each game session. A Client-Server game may have
hundreds of players connected at one moment, and the number of connected player can be thousands at another. The number of LWs is changing from time to time, however, the game always allows multiple LWs being created and simulated at the same time. The main difference between the GWG for a Client-Server game compared to the GWG for a single player game is whether the GWG of the game has multiple LWs or no LWs. The GWG framework provides support for describing multiplicity of each kind of GW. To classify GWGs according to multiplicity the following procedure is used: Once two GWGs have the same multiplicities for all three kinds of Game Worlds, we say they have the same combination of Game Worlds multiplicities. We use the term Graph Style to capture the commonality of a group of GWGs, which share the same combination. Through using 0, 1, and n to represent zero Game World, one Game World and multiple (0 to n) Game Worlds respectively, we can represent various configurations. According to the definition of the Graph Style, all possible Graph Styles should be represented in a list of all the possible combinations. This makes it always possible to find a Graph Style that complies with the combinations of Game Worlds and Connectors of a game by going through the list.

A Graph Style can be captured with the expression: \(G_nL_nP_n\), where \(G_n\) is the multiplicity of GWs (global), \(L_n\) is the multiplicity of LWs (local), and \(P_n\) is the multiplicity of PWs (perceptible). It is a trivial task to enumerate all possible Graph Styles. However, not all possible Graph Styles make sense if we consider the common understanding of video games. E.g. the 000 Graph Style means no GW, LW and PW. Such a game does not make any sense. We have defined two hypotheses that games software must adhere to in order to represent a valid Graph Styles as described below:

**H1:** A video game always has a global logical state, which can be distributed to multiple software elements, but should be complete at the system level. To our best knowledge, there is no software that is publicly accepted as a game being stateless. Also game software with only part of the global state represented is not very practical.

**H2:** A video game has to present its state to player(s) in a game specific manner. Maybe a game can use an externalization mechanism other than rasterizing and outputting on a 2D display, but it has to provide the player opportunities to perceive the game. For a pervasive game or an Alternative Reality Game (ARG), the Perceptible World may overlap with the real world, but there are still externalization mechanisms like moving an object in the real world, etc. If a software system never makes its state perceptible to the player by any means, it cannot be defined as a video game. According to the two above hypotheses, the valid Graph Styles form the following set: \(\{101, 10n, 111, 11n, 1n1, 1nn, n01, n0n, n11, n1n, nn1, nnn\}\).

The Graph Style seems to be a feasible abstraction capturing the commonalities of GWGs for creating an architecture taxonomy framework, but the information about Connectors is lost when denoting GWGs with 0, 1 or n. To address this problem, the GWG framework adds information about how the different worlds in a Graph Style are connected. The Connectors in a GWG can be zero, one or even tens of thousands, which makes it impossible to directly enumerate all Connectors in a GWG. It is therefore
necessary to abstract and simplify the amount of connections and the result has to be run-time definite in order to capture the architectural characteristics. Our solution is to use the Connector types between the three kinds of Game Worlds, e.g. if all Connectors between any particular GW and any particular LW in a game are remote Connectors, we say the GWs and the LWs are \textit{remotely} connected in the game. There are three choices of Connector types between two different kinds of inter-connected Game Worlds:

1) \textit{Entirely remote}: All Connectors connecting any two Game Worlds of different kinds are remote Connectors (Game Worlds on separate nodes).
2) \textit{Entirely native}: All Connectors connecting two Game Worlds of different kinds are native Connectors (Game Worlds on the same node).
3) \textit{Hybrid}: All situations except 1) and 2).

Two game Worlds of the same kind can also be inter-connected. This means that the above classification also works for connecting Game Worlds of the same kind. The GWG framework identifies six valid combination of connection types between game worlds: 1) Connector type of GWs with GWs, 2) Connector type of LWs with LWs, 3) Connector type of PWs with PWs, 4) Connector type of GWs with LWs, 5) Connector type of LWs with PWs, and 6) Connector type of GWs with PWs.

We define the concept Graph Flavor as the means to describe the combination of Connector type and Graph style. The Graph Flavor concept is an elaboration of the Graph Style concept based on the Connector type. E.g. one GW and one PW connected with the entirely remote Connector, and one GW and one PW connected with the entirely native Connector, can be of two different Graph Flavors, but belong to the same Graph Style (the expression is \(11\)). The Graph Flavors of the GWG can be outlined with Expression (1).

\[
G_m(c_g) c_{gl} L_m(c_l) c_{lp} P_m(c_p) c_{gp}
\]

\[
G_m(c_g) c_{gp} P_m(c_p)
\]

(When there is no LW)

\(G_m\): multiplicity of the Global World \(L_m\): multiplicity of the Local World

\(P_m\): multiplicity of the Perceptible World \(c_g\): Connector type of the GWs with GWs

\(c_p\): Connector type of the PWs with PWs

\(c_{gl}\): Connector type of the LWs with LWs

\(c_{gp}\): Connector type of the GWs with PWs

When writing an expression, the following rules must be applied:

1) Use 0, 1, \(n\) to represent multiplicities;
2) Use - to represent the entirely remote Connector;
3) Use + to represent the hybrid Connector;
For entirely native Connector, if it is

1) \( c_{gl}, c_{gp} \) or \( c_{lp} \), ignore the Connector type.

2) \( c_{g}, c_{l}, c_{p} \), use \( = \) to represent the Connector type.

Table 5.1 shows some examples of Graph Flavor expressions.

<table>
<thead>
<tr>
<th>Expression Examples</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-nn</td>
<td>One GW is remotely connected with multiple LWs, and the latter are natively connected with multiple PWs. The expression represents the GWG Flavor of a typical Client-Server game.</td>
</tr>
<tr>
<td>11</td>
<td>One GW is natively connected with one PW. The expression represents the GWG Flavor of a typical single player on single machine game.</td>
</tr>
<tr>
<td>n(-)-nn</td>
<td>Multiple GWs are inter-connected with each other remotely, and are remotely connected with multiple LWs. The latter are natively connected with multiple PWs. The expression represents the GWG Flavor of a typical multi-server, multiplayer, Client-Server game.</td>
</tr>
</tbody>
</table>

The definition of the GWG forms the foundation layer of the GWG framework, while the Graph Style and the Graph Flavor concepts form the higher abstraction layers. Based on these two concepts, we have created a taxonomy framework for classifying game architectures, which provides a new view of game architectures. The view is related to traditional architectural patterns, for example the Model-View-Controller (MVC) pattern. The GWG framework can be mapped to the MVC pattern as the GW and the LWs represent the model, the PW is the view, and the process supporting connectors represents the controller. However, the GWG is still different from MVC because it represents details on a finer granularity level where the state distribution and connections play the central roles. These issues are not covered by the MVC pattern since MVC does not (also is not intended to) differentiate between the various models, views, and controllers. The lower level of abstraction makes the GWG a taxonomy that is able to represent game architecture at a more refined level than traditional pattern based taxonomies. Moreover, the Graph Style and Flavor are useful for exploring unknown and unused game architectures, which is demonstrated in Section 5.3.4.

5.3 Use of the GWG Framework
GWG as a conceptual framework has many applications, such as:

1) It can provide conceptual level support for MDGD;
2) It provides a terminology for game architecture communication;
3) It serves a taxonomy for game architecture classification;
4) It can help analyze existing game architectures;
5) It can help explore future game architectures.

The application of GWG in MDGD domain will be illustrated by a case study in Chapter 8, and this section focuses on other applications. The use of GWG as a taxonomy is presented in Section 5.3.1, and the feasibility is validated with a systematic review of game architectures, which is described in Section 5.3.2. I also use three examples to show how to analyze game architectures with GWG in Section 5.3.3. GWG as a tool to explore future architectures is discussed and further illustrated with an example in Section 5.3.4, and the value of GWG as a terminology is implied in these applications.

5.3.1 GWG as an Architectural Taxonomy Framework

Software architectures for games vary and current taxonomies are too rough to capture the specific characteristics that make the game architectures novel. As we have discussed in Section 2.3, the GWG reflects on some essential aspects of game software architectures, and therefore it can provide some new perspectives.

Assuming that we can always identify the GWG, the Graph Style and the Flavor of every game architecture, then the architectures with the same Graph Style naturally form an architectural category. Further, a classification of game architectures can be done through identifying such categories. The classification is Graph-Style-based, i.e. each category features a specific Graph Style. Another classification based on the Flavor can also be carried out through the same process. These two classifications form two layers of the GWG taxonomy framework providing two levels of granularity. Figure 5.2 shows the framework hierarchy. In the Graph Style Layer, the Graph Style expressions like 101 and 10n are used to name the architecture kinds. While in the Flavor layer, the Flavor expressions like 1-1 and 1-nn are used to name the architecture categories.

![Figure 5.2 GWG as an Architectural Taxonomy Framework](image-url)
It is difficult to theoretically prove the assumption that GWGs can always be identified in various described architectures. It is also difficult to systematically validate the framework, since it requires a thorough review of existing game architectures, which is beyond the scope of this thesis. However, to empirically show the validity of the framework, we will analyze the architectures of three games described in the literature based on the Graph Style and the Flavor (Section 5.3.3). Before taking a closer look at these examples, we first describe the process of classifying architectures, which includes following steps:

1) Identify the three kinds of Game Worlds, and their Connectors in the game architecture, and then draw a GWG of the architecture.
2) Generalize the GWG by identifying multiplicities of the three kinds of Game Worlds, and combine the multiplicities to present the Graph Style of the architecture. The architecture can thus be fitted into the corresponding category in the upper layer of the GWG taxonomy framework.
3) Generalize the Connector types between the Game Worlds by summarizing possible Connectors between combinations of the Game Worlds. Combine the information of the Connector types with the Graph Style, and the elaborated Graph Style represents the Graph Flavor of the architecture. The architecture can thus be fitted into the corresponding category in the lowest layer of the GWG taxonomy framework.

6.3.2 A Systematic Review Based on the GWG Framework

A systematic review of game architectures in literature was carried out to validate the usefulness of GWG as a taxonomy framework. Four electronic databases were searched during the review, and 40 articles describing game architecture were included according to the inclusion criteria. The architectures described in the articles were analyzed from the GWG perspective, and then fitted into the GWG taxonomy framework. The architectures ranged from single player games to multiplayer networked games, and the architectures are representative for the field to our best knowledge. Our hypothesis was that the architecture of the games found in the research literature would only fit into one of the Graph Style categories, and into only one of the Flavor categories. The systematic review method used was adapted from [106], and it was tailored according to the domain. The complete review is presented in Chapter 6, and I briefly describe the results in this section.

Figure 5.3 illustrated the classification according to Graph Style. All the possible GWG Graph Styles are enumerated and the number of architectures found for each Graph Style is reported.
The results showed that the game architectures found in research literature are centered on the Graph Styles 1nn, n0n and nnn, and the latter gets the most attention. There were also a few papers that described game architectures with the 101 and 10n Graph Styles. Another observation was that 7 categories of Graph Styles are not present in research articles, showing that the research on game architectures focus only on a limited set of Graph Styles. The following reasons may explain the focus on few Graph Styles:

1) Research on game architectures focus on proposing innovative software architectures and not relatively mature and standard architectures such as architectures for stand-alone games (no network support).

2) Not all of the Graph Styles are practical. For example n1n Graph Style implies multiple Perceptible Worlds with one Local World, which describes a single player-multi-view game. However, it is impractical to have multiple Global Worlds to support such a game style.

3) The Graph Style captures important attributes of game architectures, but the classification is still at a high conceptual level. This means that there can be several differences between software architectures within the same Graph Style. GWG Flavor can be used to capture some of these differences on a lower abstraction level.

Figure 5.4 showed the classification according to the GWG Flavors. If a GWG Flavor was present in an article, a Flavor sub-category in the corresponding GWG Graph Style category was added to Figure 5.4.
Figure 5.4 Classification According to GWG Flavors

Figure 5.4 shows that the 40 architectures are categorized, and the distribution is fairly even, although the categories $n(-)n$, $n-nn$ and $n(-)-nn$ are relatively concentrated. These three architectural styles characterize two kinds of Multi-Server architectures and a Peer-to-Peer architecture. This observation adheres to our knowledge about the focus of game architecture research. There are still more possible Graph Flavors that are not present in the reviewed articles, and some of them reveal potential for new architecture innovation. This will be discussed in Section 5.3.4, and Section 5.4 describes a validation of the inferential power of the GWG framework.

The next section describes and analyses three game architectures selected from Figure 5.3. The examples are picked from the three Graph Style categories that are most commonly described in the literature. Thus, these examples reflect the use of GWG on the architecture styles game researchers are most interested in.

5.3.3 Analyze Three Game Architectures with GWG

The examples of game architectures were selected from the top three most popular categories identified in the review. The categories are $1nn$, $n0n$, and $nnn$. The architectures represented in these three categories are Client-Server, Peer-to-Peer, and Multi-server architectures respectively.

Game architecture 1: Strifeshadow Fantasy (SSF) [107] is a massive multiplayer online role-playing game with over 10,000 registered players. In SSF, a central server hosts the SSF server program simulating the entire Game World. Players connect to the server through the SSF client application. The HTTP protocol is used for client-server communication, and Macromedia Flash 4.0 is used to render the game scene and regularly sends Update Request Message (URM) to the server. The server is based on Active Server Page (ASP) combined with HTML, scripts, and Microsoft ActiveX server
components. The server uses a relational database management system to store and retrieve the game state. In SSF, a unique Global World is maintained on the server while each player connected to the server has a client managing a Local World on his/her device. The Perceptible Worlds are— also presented on players’ devices. Figure 5.5 shows the GWG of the SSF game.

![Figure 5.5 The GWG of Strifeshadow Fantasy](image)

The multiplicity of the Global World of SSF is “1”, while the Local and the Perceptible Worlds are “n”. This means that SSF falls into the Graph Style 1nn category in the GWG Framework. The Connector type between the Global and the Local Worlds is entirely remote while the Connector type between the Local and the Perceptible Worlds is entirely native. This means that the SFF falls into the 1-nn Flavor category.

**Game architecture 2: VoroGame** [108] uses a structured P2P overlay based on a distributed hash table (DHT) (e.g.[109, 110]) combined with an unstructured P2P overlay based on a Voronoi diagram [111] to manage the global state. The Game World is divided into zones based on the Voronoi diagram, and DHT is used to store game objects on peers evenly and fairly. Each peer has two roles in the architecture 1) **Voronoi responsible** related to a zone in the Voronoi diagram which is in charge of simulating the behavior of objects inside the zone that the player of the peer locates in, and 2) **DHT responsible** that is in charge of storing objects and forwarding objects state changes from Voronoi responsible to players that require information about the objects state. Figure 6 shows an example of the GWG of the VoroGame. Note that the number of players changes dynamically, so there can be various GWGs for this architecture. However, the game will be categorized in only one Graph Style and one GWG Flavor.
The multiplicity of the Global and the Perceptible Worlds of the VoroGame are “n”. The Local World is not presented in the architecture. So the Graph Style of GWG of the VoroGame is n0n. The Connector type between the Global and the Perceptible Worlds is entirely native, and the Connector type between the Global Worlds is entirely remote. This means that the VoroGame falls into the n(-)n Flavor category.

**Game architecture 3: Rokkatan** [112] is a multiplayer online game based on a Mirrored Server architecture. The game state is simulated on multiple servers, and every server manages a full instance of the entire game state. The servers use a cooperation protocol to keep the game state consistent. Players joining the game can select the fastest server to get a relatively smooth gaming experience. Players interact with the Global World through their client machines where Local Worlds are simulated and the game is presented to the user (Perceptible World). Figure 7 shows an example the GWG of Rokkatan. Note that there may be various ways of organizing servers and thus corresponding variations of Global Worlds structures.
The multiplicity of the Global Worlds of Rokkatan is “n”, and that of the Local Worlds and the Perceptible Worlds are also “n”. This means that the Rokkatan game falls into the nnn Graph Style category. The Connector type between the Global Worlds is entirely remote. The Connector type between the Global and the Local Worlds is also entirely remote, while the Connector type between the Local and the Perceptible Worlds is entirely native. This means that the Rokkatan game falls into the n(\text{-})-nn Flavor category.

5.3.4 Use the GWG framework to Explore Future Architectures

Apart from classifying existing game architectures, the GWG framework can also help in exploring future architectures. A general process of this application is described below:

1) Examine the distribution of existing architectures in the GWG categories (Graph Style based) and sub-categories (Flavor based), and identify the unfilled categories/sub-categories.

2) Each unfilled category/sub-category describes a potential high-level architecture pattern (only architectural characteristics related to the GWG are described). By examining the Graph Style/Flavor of the category/sub-category, we can reason about the major features of the architecture pattern.

3) Matching the features of the architecture pattern to the architectural drivers [86] connected to each game type to justify if these characteristics are appropriate for a specific game type.
4) If an architecture pattern is considered appropriate to a game type, add more architectural tactics that are relevant to the game type to make it a concrete architecture.
5) Evaluate the architecture with either empirical or theoretical methods.

1) and 2) are directly supported by the GWG framework itself, while 3) to 5) require cooperation with other architectural methods. This means that the role of the GWG framework in this innovative process is to be a tool for generating the new high-level architecture patterns. We use an example to illustrate how the GWG can be applied in such a way in the rest of the section.

Through analyzing Figure 5.3 and Figure 5.4, we find that in the Graph Style Inn category, there are four GWG Flavors identified in existing architectures, which are 1-nn, 1-n(-)n, 1-n-n and 1-n-n-. It is a trivial task to enumerate the possible GWG Flavors in the Graph Style that are not featured by any existing architecture, for instance, 1-n+n, 1-n(-)n-, and etc. We will further explore 1-n+n as an illustration.

The 1-n+n Flavor implies that multiple players interact with a single, shared Game World (one GW) through their individual views (multiple PWs), and the Remote Connector type between the GW and the LWs reveals that the GW is simulated on an individual node. Since the PWs are always presented on the machines players directly interact with, the hybrid Connector between the LWs and the PWs implies two kinds of player machines, one doing local state simulation, rendering and presentation, and the other (thin clients) only doing presentation. The thin clients connect to the rendering servers through a network, and the rendering servers maintain the local game state (LWs) and stream the rendering results to the thin clients. The characteristics of the architecture pattern implied by the 1-n+n Flavor can be summarized as follows:

1) Single centralized machine simulates the global state.
2) Local state is maintained on multiple machines for rendering and player interaction.
3) The above two kinds of machines are connected through a network.
4) Players interact with two kinds of machines, one is a fully-functioning client machine in charge of local state simulation, rendering and presentation, and the other is a thin client machine only in charge of presentation.
5) Remote rendering is done on rendering servers, which maintain their own local game state. The game server is dedicated to the global state simulation.

The characteristics 1)- 3) are the same as traditional centralized server architectures, but the characteristics 4) and 5) make the architecture pattern novel. These characteristics describe that local and remote rendering are supported at the same time, meaning that client devices with different computational capabilities can be used. The remote rendering is done on rendering servers, not on the GW server, which differentiates the architecture pattern from its 1-nn- siblings. The architecture is appropriate for mobile ubiquitous games, which run on heterogeneous devices. Further, if we use some of the clients that are capable of rendering as the rendering servers for the thin clients, the
architecture will have the advantages of Public Server architectures, thus can provide better scalability and save investments on rendering servers.

As a summary, the architecture pattern can be elaborated by defining three kinds of client devices: i) Fully-Functioning Client (FFC), ii) Thin Client (TC) and iii) Super Client (SC). Figure 5.8 shows the architecture overview.

**Figure 5.8 A Hybrid Remote-Rendering Architecture with 1-n+n GWG Flavor**

In Figure 5.8 we see that high performance smart phones are used as the FFCs, desktop PCs are used as SCs and low-end cellphones are used as TCs. Each TC is connected to a SC, which generates visual game scene as graphics/video stream for it. The central game server delivers state updates to FFCs and SCs. The FFCs having full graphics capabilities, maintain their local state and render the game world. The SCs are more powerful, thus they maintain not only the game state for their direct users but also the game state for the connected TC users. The SCs render the Game World for their users and generate graphics/video streams for TC users in parallel. The TCs do not directly process the state update from the central game server, and they connect to SCs to fetch the streamed scene for playback. For players’ inputs, FFCs and SCs both directly handle player input and communicate with the central game server, while TCs only forward the input to the connected SCs since the input handling needs the local state as context. To make it simple, the data stream from the clients to the server is not presented in Figure 5.8.

**5.4 Quality of GWG**

This section will discuss and analyze the quality of the GWG framework. The term *quality* means the appropriateness and the usefulness of using the GWG framework for
understanding game architectures from a pragmatic perspective. The quality evaluation is analysis-based by adapting a framework for evaluating theories from the Computer Supported Collaborative Work (CSCW) domain. In addition to analyzing the quality of the framework, practical experiences and issues from using the GWG to analyze architectures of games in general will be presented.

In [113], Halverson proposes a framework for evaluating a theory from pragmatic view. The framework is used to evaluate theories in the CSCW domain. However, it can also be adopted to evaluate the concepts and foundation of GWG, since the framework is general in concept. In Halverson’s framework, four attributes of theory are identified which are required by the people who use it. The attributes are: 1) Descriptive power – Theory should provide a conceptual framework that helps us make sense of and describe the world; 2) Rhetorical power – Theory should help us talk about the world by naming important aspects of the conceptual structure and how it maps to the real world; 3) Inferential power – Theory should help us make inferences; and 4) Application - How we can apply the theory to the real world for essentially pragmatic reasons.

In the GWG framework, Game World and the World Connector are defined as the basic concepts, building the formalized foundation for the concept of Game Worlds Graphs. The GWG reflects some essential characteristics on how game software is organized. The GWG can be regarded as a means to describe the game architecture domain. Moreover, the Graph Style and the Graph Flavor are proposed as abstractions of the basic GWG concepts, which describe the game architecture at a higher abstraction level. The Graph Style and the Graph Flavor serve as the cornerstones of the taxonomy framework, which describe the commonalities and the differences of game architectures. The framework is sufficiently descriptive to not only describe variations of the architectures of networked games as most of the taxonomies can do (but at a higher abstraction level), but it also categorizes the architectures of single player games. The validity of the GWG as a taxonomy framework is supported through a systematic review where some of the relevant results are provided in Section 5.3.2. Further, Section 5.3.3 shows how the GWG framework can be used to analyze three different game architectures and how our framework highlights their differences. The GWG framework can describe the game architectures as well as the classification of the game architectures. The above arguments demonstrate the descriptive power of the GWG framework.

The GWG framework provides a terminology for naming the Graph Style and the Flavor. The terminology naturally works for naming the categories in the GWG taxonomy framework, since the categories are based on the Graph Style and the Flavor. The terminology includes two layers:

1) Graph Style layer, where expressions like 101, 10n are used to denote the architectures with information on Game Worlds distribution.

2) Flavor layer, where more complicated expressions like 1-1, 1-nn are used to denote the architectures with information on both Game Worlds distribution and Connector types.
The expressions are informative and efficient since essential characteristics of game architectures (Game Worlds distribution and Connector types) are presented through simple expressions. The expressions are intuitively more concise and expressive than existing terminology (e.g. Client-Server, Peer to Peer and Hybrid), as they reveal important architectural differences that architectural pattern names currently do not capture. The expressions are layered so they can provide two levels of granularities allowing them to be more adaptive to wider domains. The capability of the terminology in naming phenomena in game architecture shows the *rhetorical* power of the GWG theory.

The GWG framework can help reason about quality attributes of architectures, when they are fitted into the framework. E.g. for n(-)-nn architectures, the duplication of the Global World can usually avoid single point of failure, which results in better availability than 1-nn architectures. Sometime the inferences can even be about the characteristics of unknown architectures. In Section 5.3.4 we presented an example of using the GWG framework to explore a Hybrid Remote-Rendering architecture, which to our best knowledge has not been proposed in any previous research article. Although the described architecture still needs to be refined to solve issues such as incentive mechanism and cheat prevention, the exploration has provided an architecture design prototype. More proposals of new game architectures can be made using the GWG framework. E.g. in the n0n category, we can reason with the GWG framework that possible flavours may include n-n, n(-)-n, n(-)n, and so forth. However, only n(-)n architectures are presented in articles we reviewed. With the analysis based on the GWG, we foresee that the n(-)-n architecture can be a derivation of the Mirrored Server Architecture, where the Remote Rendering techniques are used to remove rendering computation from the client. I believe such *explorations* will result in proposals of new architecture patterns that can be useful and reveal new opportunities. A general approach of using the GWG to explore unknown architectures is to enumerate possible Graph Styles and Flavors, and identifying categories that make sense but that have not yet been implemented in games. Thus, new architectures based on identified GWG Graph Styles and Flavors can be defined and explored. The GWG framework opens for exploration into new territories of existing and future architectures and which proves the *inferential* power of the GWG framework.

When it comes to *Application*, I have identified five opportunities in Section 5.6.3: support MDGD, architecture terminology and taxonomy, analyzing exploring architectures.

### 5.5 GWG Composite

Games today often provide different modes in how players play the game that includes both single player, multi-player on same machine and multi-player-online, which are implemented using different architecture patterns. An example is Ubisoft’s action game Splinter Cell: Double Agent, which has a single player mode and a multiplayer mode. In the single player mode, player unfolds a story by completing missions, and the architecture is the typical stand-alone architecture, which falls into the 11 category. In the multiplayer mode, players use client machines with network connection and compete against each other by achieving predefined goals, and the architecture is a
Client-Server architecture, which falls into the 1-nn flavor. The example reveals that sometimes it is not possible to describe the architecture of a game only with one architectural pattern. In other words, some games can hardly be regarded as one game from the architectural point of view as they feature totally different game modes. We define the term *composited architecture* to denote this phenomenon. The term *GWG composite* captures the characteristic of the GWGs for composited architectures in one game. A GWG composite can consist of two or more GWG Graph Styles and Flavors, e.g. 11 + 1nn and 11 + n-nn.

Understanding of GWG composite can help us investigate the game architecture from a higher level of abstraction. When we technically study the architecture of a game, we tend to simplify design constraints and user demands on gameplay, thus we generalize the architectures with abstract patterns like Client-Server, Peer to Peer, etc. However, when developing a game, the composite architectures can usually help in building practical insight. The GWG composite is capable of describing the composited architecture, and the GWG composite can also help analyzing the appropriateness of different composited architectures before the design decision is made. Further analysis of GWG composite of several existing games can also be used to identify if there are technical challenges in integrating various GWG Styles and Flavors in one game, and possible solutions for dealing with such integration. This is another unexplored area of research initiated by the work on the GWG framework.

### 5.6 Comparison with Existing Taxonomies: The Related Work

Most of the taxonomies in literature focus on networked games and the categories are defined in an ad-hoc way. In [95], architecture considerations on MOGs were discussed, where game architectures were summarized as i) Client-Server, ii) Peer-to-Peer, and iii) Hybrid architectures. The author used the term “Hybrid” to describe the Mirrored Server Architecture. This taxonomy does not cover games other than networked games, and it uses the “Hybrid” category, which is an ambiguous term, to cover any variant that does not fall into the Client-Server or Peer-to-Peer categories. For instance, different architectures were named “Hybrid” in [94-96]. Similarly, in [114], the game architectures were classified into three categories: i) Client-Server, ii) Peer-to-Peer, and iii) Federated Peer-to-Peer. The latter is a Peer-to-Peer architecture with group management, where players in the same interaction group form a multicast group, and Application Level Multicast (ALM) is used for communicating within the group. This taxonomy does not cover non-networked games, and does not cover Multi-Server architectures.

Another taxonomy was introduced in [93], where four categories were defined: i) Peer-to-Peer, ii) Client-Server, iii) Mirrored-Server, and iv) Clustered-Server. In this taxonomy, multi-server zone-based architectures and mirrored server architectures were distinguished. A more elaborated taxonomy was proposed in [115], with five categories defined: i) Client-Server, ii) Peer-to-Peer, iii) Client-Multi-Server, iv) Peer-to-Peer with Central Arbiter, and v) Mirrored-Arbiter. The GWG is different from these taxonomies at least for three reasons:
1) The GWG is a taxonomy framework intending to cover all game architectures, which is not restricted to only networked games.

2) The GWG is a systematic framework with inferential power.

3) The GWG provides a more informative, unambiguous and efficient terminology.

The taxonomy proposed in [116] is a framework that is the one closest to the GWG. The framework also uses the graph concept and defines the term communication graph. However, for communication graphs, the node is the computing node and the edge is the network communication. This framework is different from the GWG framework in the following points:

1) The node in the GWG framework is defined with the Game World, while the node in the communication graph is defined with the computing node. So the GWG framework focuses on the state distribution, and the communication graph focuses on the computation distribution.

2) The edge in the GWG framework describes the synchronization between the Game Worlds, while the edge in the communication graph describes the network communication. The synchronization is more general than network communication, which can be the network communication but is not restricted to network communication.

3) The GWG identifies Perceptible World and is able to describe how the user interface is related to the rest of the game software. It can therefore describe a wider variety of architectures such as remote rendering architectures.

4) The GWG framework defines Graph Style and Flavor as the abstraction of commonalities of GWGs. These concepts can help infer attributes of existing architectures and build insight into future architectures.

5) The GWG framework provides an informative terminology to name categories.

In [26], a taxonomy is introduced to characterize pervasive games/applications. This taxonomy focuses on abstracting application characteristics, independent of the characteristics of the middleware or infrastructure that support the application, and provides a controlled vocabulary for thinking about the application. These characteristics may impact the application architecture. The taxonomy is not directly about architectures, but it can be used in framing discussions about potential designs and architectures of applications. We think the potential synergy of using the taxonomy with GWG in exploring future architectures is worth further studies.

5.7 Summary

To support various game architectures, MDGD requires a conceptual base that is able to guide the DSLs design and code generation. GWG is a framework that was created for this purpose. However, as a conceptual framework the application domain of GWG is much broader than MDGD, and in this chapter I mainly discussed how it is useful in classifying, analyzing and exploring game architectures. Due to the focus consideration, the application of GWG in MDGD is illustrated with a case study in Chapter 7.
The GWG is based on the Game World and the Connector concepts. The framework provides a new perspective on game architectures and reveals some essential characteristics of game architectures, thus the framework is useful for getting a deeper understanding of game architectures. The Graph Style and Flavor are two concepts to capture commonalities of GWGs, and a taxonomy framework for game architectures is defined based on these concepts. To name the categories in the taxonomy, a terminology is proposed, which is more informative, descriptive and efficient than existing ones. GWG provides new terminology for discussing and categorizing software architectures without the need of explaining all the details of the architectural patterns used. The validity of the taxonomy framework is backed up by being able to distinctly classifying 40 game architectures described in the literature from 1990 to 2009 more elaborate than done before, and as well showing the validity through a detailed analysis of three game architectures.

Apart from being a tool for analyzing existing architectures, the GWG framework can be used to explore future architectures. The Graph Styles and Flavors that are not identified in known architectures provide hints on potential architectural patterns, and can thus help create and analyze novel architectures for modern games and even future games. For instance, the GWG framework is capable of analyzing various cloud-based architecture for games by considering various configurations for how the game state is distributed in the cloud and on the clients, as well as how the presentation of the game is handled (streaming vs. client presentation). The quality of the GWG was further evaluated based on an evaluation framework adapted from the CSCW domain. The evaluation argues for the descriptive power, the rhetorical power, the inferential power and the application of the GWG framework. Furthermore, the GWG composite concept is included to address the problem when a game consists of several game modes with different architectural characteristics.

Computer games evolve with the change of technical, social and cultural context, and the GWG framework also needs to be updated continuously. The most interesting improvements of the framework we can foresee include identifying more kinds of Game Worlds and Connectors, and adding concepts that can make the framework more powerful. Future work includes developing and evaluating concrete new architectures that are revealed by the GWG. For improving the framework, the update flows through connectors should be further researched.

I will present the results from the literature review based on GWG in next section, which not only validated the usefulness of GWG, but also supported the MDGD industrialization with providing valuable information of the game architecture domain.
6 From 101 to NNN: The Diversity of Game Software Architectures

MDGD has to consider the diversity of game software architecture in order to be useful in commercial game development, because it influences many aspects of MDGD, such as DSLs, code generators and domain frameworks. There are many different game architectures today, and the Game Worlds Graph (GWG) framework provides a conceptual base, which can not only support MDGD but also help analyze and classify computer game architectures.

This chapter presents a systematic review of game architectures using the GWG framework. The review validates the usefulness of the GWG framework through classifying game architectures described in the literature into distinct categories according to the framework. The major contribution of this chapter is a state-of-the-art presentation of 40 different game architectures, which covers architectures for all kinds of games from single player games to Massively Multiplayer Online Games (MMOGs). Previous reviews of game architectures have focused on a narrower selection of games such as only networked games, MMOGs or similar. Further, none of the previous reviews has used a systematic framework for analyzing the characteristics of game architectures. By using the framework, the similarities and differences of the 40 game architectures can be identified in a systematic way.

The review is a domain investigation of game architecture, which provides useful information for designing the practical MDGD approaches that support diversified game architectures.

6.1 Review the Game Architectures
The software architecture of a computer game influences both the quality attributes and the functionality, and has therefore played an important role in game development. MDGD has to consider the diversity of game software architecture in order to be useful in commercial game development, because it also influences many aspects of MDGD: The DSL developer must have the target architecture in mind; the code generator should be able to generate the infrastructure code for the target architecture; the domain framework has to support the target architecture.
Today, game designers and programmers have the opportunities to create new social and innovative games using technologies like the Internet and hardware and services provided by mobile computing, which drives the evolution of game architectures. Game architectures are getting larger, more complex and more sophisticated, evolving from stand-alone to Massively Multiplayer Online Game (MMOG) architectures. Creating an practical MDGD approach requires deep understanding on the game architecture domain, in order to make proper architectural decisions when developing the DSL, code generator, domain framework and so on. Since the domain knowledge is an important input to MDGD, a thorough investigation on the architecture is carried out in my Ph.D. research for building the knowledge, and the results are reported in this chapter, which are based on a literature review including 40 game architectures that are fitted into the GWG framework.

The second contribution of the literature review is that the appropriateness of GWG as a taxonomy framework for game architecture is validated. The expansion and higher complexity of architectures introduce new challenges such as state consistency, anti-cheating, load balancing. Such issues make it important to have a descriptive taxonomy framework, which is capable of classifying different architectures in sufficient detail. The Game Worlds Graph (GWG) framework introduced in Chapter 6 is a suitable choice, which is a systematic framework capable of not only classifying and analyzing existing architectures but also exploring potentially new and unknown architectures. The results presented in this chapter have been achieved by conducting a systematic review of research articles, where game architectures described in these articles are analyzed, and then categorized according to the GWG framework. Each identified architectural category is discussed along with advantages and challenges of this category based on its GWG attributes. The major architectural characteristics of each reviewed architecture are also presented.

The related work in literature is restricted to one specific domain of games with emphasis on network games or more specifically MMOGs. Systematic review as a research method has not been used before on reviews of game architectures. In [117], Jiang et al. have reviewed software architectures of network games. They classified architectures in three categories: Distributed Server Architecture, Hybrid Peer-to-Peer Architecture, and Structured Peer-to-Peer Architecture. The article discusses the characteristics of architectures with the focus on design tactics to provide scalability. The review has a narrower scope and a more specific focus than our work. [118] is another review article which focuses on techniques for Peer-to-Peer MMOGs. The authors mainly use six design issues related to Peer-to-Peer MMOGs as their framework with which design tactics are discussed. Since these tactics are elements of software architecture, the review is performed at a finer granularity. To illustrate the design tactics, six representative architectures are analyzed, which are all Peer-to-Peer architectures. Compared to the related work described above, my literature review is different in at least three aspects:

1) Systematic review was used as the research method, which results in a better coverage of the literature.
2) The GWG framework was used as the review framework, which categorizes game architectures according to Game World decomposition and connections. The framework provides an approach for characterizing game architectures found in the literature according to main characteristics, without getting into low-level technical details.

3) Game architectures for all kinds of computer games were included in the review, which is not restricted to one type of games or one type of architectures.

Based on the arguments above, the systematic review draws a more complete picture of game architectures covering innovations on architectures of all kinds of computer games, from stand-alone games to MMOGs, which has not been presented in any previous articles in the literature.

6.2 Research Method and Design

I adapted the systematic review method used in [106] in the research, which organizes the review through five distinct stages: development of review protocol, identification of inclusion and exclusion criteria, search for relevant studies, critical appraisal, data extraction, and synthesis.

Since the purpose of the review was to review and classify existing architectures, quality evaluation of the architectures was irrelevant. This means that the critical appraisal stage could be skipped. Therefore the review was conducted according to these four stages:

1) The development of review protocol;
2) The identification of inclusion and exclusion criteria;
3) A search for relevant studies;
4) Solutions’ extraction and classification.

These stages are described in detail in the remaining of this section.

6.2.1 Protocol Development

The protocol used similar to the one in [106], and it was tailored through a dedicated discussion among the authors. This protocol specifies the search strategy, inclusion and exclusion criteria, method of solutions extraction and classification.

6.2.2 Inclusion and Exclusion Criteria

The purpose of the research was to review and classify game architectures in literature. Articles that described one or more architectures were included. This included two types of articles:

1) Articles that claimed to propose one or more innovative architecture solutions.
2) Articles that described one or more architectures in sufficient detail, but the architectures had been proposed in an earlier published article.
Work of both researchers and practitioners were included. The review included articles from 1990, and up to and including 2009 (since I searched the databases in 2010, articles published in 2010 and later were not included). Only work written in English was included. The architectures that were included in the review had to be architectural descriptions that covered all main parts of the software necessary for representing a computer game.

Articles were excluded if they did not provide sufficient details of the described architecture(s) to analyze them. Articles were excluded if they only presented a partial architecture of game software, e.g. rendering engine architecture, client architecture. Articles were excluded if they presented an architecture that included system(s) other than the gaming system itself, e.g. payment system, user management system, news system, etc. Articles were excluded if they described an architecture that had previously been proposed in another article, while the latter had been included in the review (to avoid duplicated architectures).

6.2.3 Search for relevant studies

Due to time and resource constraints, only electronic databases were searched. The following four electronic databases were selected according to my knowledge of the field and the time budget:

- ACM Digital Library
- Compendex
- IEEE Xplore
- ScienceDirect – Elsevier

The following keywords were used:

(Game OR Gaming) AND Architecture

“Game” and “Gaming System” are two popular names for game software. “Architecture” can mean different concepts such as building and network architectures, but there is no other keyword that is more specific. The keywords were searched in titles, abstracts and indexes. Four databases were searched, resulting in a total of 2766 “hits”. These findings were going through and irrelevant articles were excluded by looking at the title, the publication source, and the abstract. The papers that had not been excluded were imported to the EndNote bibliography tool. Duplicated papers were excluded. A total of 154 articles were considered as potentially relevant after this step. All 154 candidates had to be examined carefully by studying the full texts of the articles, excluding articles according to the inclusion and exclusion criteria, and a total of 40 articles were selected for reviewing.

6.2.4 Architecture Solutions Extracting and Classification

For this stage a table with all 14 valid Graph Styles was created forming 14 architectural categories. Then the game architectures described in the 40 articles were analyzed and classified into these 14 categories in the table. Further, the architectures were analyzed.
according to the GWG Flavors, and placed into corresponding sub-categories. If a sub-category did not exist from previous analysis, a new sub-category for the GWG Flavor was created and the architecture was placed into this sub-category.

After this characterization process was finalized, a tabular view of the architectures was produced describing the reviewed articles categorized according to Graph Styles and Flavors. The next step was to further analyze and discuss the reviewed architectures in relation to their Graph Styles and GWG Flavors. This analysis resulted in a summary of the challenges and advantages connected to each category and sub-category according to my knowledge and existing literature. This not only shows the reasoning power of the GWG framework, but also gives the big picture of game architecture research. The review results are reported in next section.

### 6.3 The Review Result

The 40 reviewed architectures fall into 5 Graph Styles, and 13 GWG Flavors. Table 6.1 shows a tabular view of how the architectures are distributed according to the Graph Style and Flavor, and these architectures are represented with the corresponding sources.

<table>
<thead>
<tr>
<th>Graph Style</th>
<th>GWG Flavor</th>
<th>Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1-1</td>
<td>[119]</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>[120]</td>
</tr>
<tr>
<td>10n</td>
<td>1-n</td>
<td>[121]</td>
</tr>
<tr>
<td></td>
<td>1n</td>
<td>[122]</td>
</tr>
<tr>
<td>11n / 11n / 1n1</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>1nn</td>
<td>1-nn</td>
<td>[97], [107], [123]</td>
</tr>
<tr>
<td></td>
<td>1-n-n</td>
<td>[98]</td>
</tr>
<tr>
<td></td>
<td>1-n-n</td>
<td>[124]</td>
</tr>
<tr>
<td></td>
<td>1-n(-)n</td>
<td>[125], [99]</td>
</tr>
<tr>
<td>n01</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>n0n</td>
<td>n(-)n</td>
<td>[126], [127], [128], [103], [108], [129], [130], [131], [132]</td>
</tr>
<tr>
<td>n11 / n1n / nn1</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>mnn</td>
<td>n-nn</td>
<td>[133], [134], [135], [136], [137], [93], [138], [139], [140], [141]</td>
</tr>
<tr>
<td></td>
<td>n(-)-nn</td>
<td>[142], [143], [144], [145], [102], [146]</td>
</tr>
<tr>
<td></td>
<td>n-n(-)n</td>
<td>[147], [115], [148]</td>
</tr>
<tr>
<td></td>
<td>n(-)-n(-)n</td>
<td>[149]</td>
</tr>
</tbody>
</table>

From Table 6.1 we can see that most of the architectures fall into 1nn, n0n and mnn Graph Styles and n(-)-nn, n(-)n, and n-nn Flavors. Note that among the 40 architectures, 38 architectures fall into GWG Flavors with at least one remote Connector Type. This
shows that the game architecture research mainly focuses on network game architectures.

To make the Flavor expressions easier to understand, the major architectural characteristics of each GWG Flavor was analyzed, and an alias system was proposed as shown in Table 6.2, where the terms used in the reviewed articles were also summarized. These terms are usually based on names for architectural patterns, e.g. Client-Server, Peer-to-Peer.

<table>
<thead>
<tr>
<th>GWG Flavor</th>
<th>Flavor Alias</th>
<th>Terms Used in Reviewed Articles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Remote Rendering</td>
<td>“Streaming Architecture”</td>
</tr>
<tr>
<td>11</td>
<td>Single Software</td>
<td>NA</td>
</tr>
<tr>
<td>1-n</td>
<td>Multi-Client Remote Rendering</td>
<td>NA</td>
</tr>
<tr>
<td>1n</td>
<td>Single Software Multiple View</td>
<td>“Parallel Rendering Architecture”</td>
</tr>
<tr>
<td>1-nn</td>
<td>Client-Server</td>
<td>“Centralized Server”, “Server-Client”</td>
</tr>
<tr>
<td>1-n-n</td>
<td>Remote Rendering Server</td>
<td>“Client-Server”</td>
</tr>
<tr>
<td>1-n(-)n</td>
<td>Remote/Local Rendering Client-Server</td>
<td>“Client-Server”</td>
</tr>
<tr>
<td>1-n(-)n</td>
<td>Client-Server with Client Communication</td>
<td>“Hybrid”, “Peer-to-Peer with Central Arbiter (PP-CA)”</td>
</tr>
<tr>
<td>n(-)n</td>
<td>Peer-to-Peer</td>
<td>“Peer-to-Peer”, “Hybrid”, “Federated Peer-to-Peer”, “Hybrid P2P”</td>
</tr>
<tr>
<td>n-nn</td>
<td>Multi-Server Partitioning</td>
<td>“Multiple-Server”, “Peer-to-Peer”, “Distributed”, “Clustered-Server”</td>
</tr>
<tr>
<td>n-n(-)n</td>
<td>Multi-Server Partitioning with Client Communication</td>
<td>“Mirrored-Arbiter (MA)”, “Hybrid Peer-to-Peer”</td>
</tr>
<tr>
<td>n(-)-n(-)n</td>
<td>Mirrored Multi-Server with Client Communication</td>
<td>“Enhanced Mirrored-Server”</td>
</tr>
</tbody>
</table>
We see that some terms such as “Hybrid”, “Client-Server”, and “Peer-to-Peer” are frequently used although they do not distinguish between architectures with important different characteristics. E.g., “Client-Server” (including “Server-Client”) is used to label various architectures spanning over 3 GWG categories from classical client-server architectures to remote rendering architectures. Other terms such as “TB-SMC” contain too detailed information about one architecture, so that they are restricted to one or very few architectures - thus become useless for classification purposes. Finally, some terms like “Hybrid” are not sufficiently informative thus readers cannot understand the major characteristics of the architectures when the terms are used.

The weaknesses of the traditional terminology for classifying game architectures highlight the contribution of the GWG framework both as a classification tool and for defining a more expressive and concise terminology. The remaining subsections discuss the game architectures in the review in detail in sequence according to the order of GWG categories as given in Table 6.1.

6.3.1 Graph Style 101 Architectures

The 101 category describes architectures consisting of one GW, and one PW. This does not necessarily indicate a single player game, but then the players must be located at the same place in order to interact with the single PW. 101 architectures can hardly be found in literature. One possible explanation is that this Graph Style usually represents traditional and trivial architectures, and thus not very interesting to researchers.

When a single GW and a single PW are natively connected, it belongs to 1/1 Flavor, which we have named Single Software architectures. Such architectures integrate all gaming processes and data into a single piece of software, where the simulation (implied by the GW) and the presentation (implied by the PW) are the two major computational tasks. The presentation of a game is directly generated from the game state simulated within the same running instance of the game. Although there is no network involved, the architectures still can be complicated, because the simulation might involve techniques like AI, physics, and presentation might involve techniques like texture mapping and lighting that are difficult to carry out in real time. Game engines that implement such techniques are widely used in commercial game development and fit within the 1/1 Flavor. Although the history of Single Software architectures is as long as the history of the computer games, today such architectures are still widely used in popular video games such as New Super Mario Bros Wii (Figure 6.1), Rayman Ravin Rabbits (Figure 6.2) and etc. Sollenberger and Singh present an architecture for affective social games in [120], which is the only architecture we found in our review that can be fitted into this category.
Another *101* Flavor identified in the review is the *1-1*. The remote Connector Type between the GW and the PW implies data flow across OS processes (usually through network). *1-1* architectures split simulation and presentation/interaction, so the task of each computational node is simplified. Thin clients with only graphics or text-based rendering and input handling capabilities are supported by such architectures. The simplification of the client software also decouples the client from some of the modules of the game engine, making it easier to provide support for cross-platform games.
(portability). Powerful computers can be used as servers to simulate multiple game
sessions on the same machine, which optimizes the resource utilization by allocating
available resource for other game sessions. In [119], Nave et al. present an architecture
falling into 1-1 Flavor. The architecture deploys simulation and rendering on a server,
while streaming the rendering result to a thin client which is only responsible for the on-
screen presentation and user inputs management.

6.3.2 Graph Style 10n Architectures

Similar to 101 architectures, architectures that fit into the 10n pattern are also widely
used in commercial games but are rarely found in the literature. By extending a single
PW to multiple PWs, 10n architectures provide support for multiplayer, multi-view
games. The simplest case of 10n is the In Flavor, where the GW and the PW are native
connected implying a multiplayer game without the need for network connectivity.
Racing games are games that commonly utilize In architectures, e.g. Mario Kart (Figure
6.3) and Ridge Racer (Figure 6.4). Such games run on a single console connected to a
single screen (TV), while multiple players can play through a split-screen that provides
a separate view for each player. The rendering has to be done multiple times in the split-
screen mode, which often cause performance challenges in order to provide smooth
frame-rates with the same graphical fidelity. In [122], Lages et al. present a parallel
architecture for multiple view rendering on GPU clusters which is appropriate for
creating In game architectures. The main idea is to combine sampler nodes with image-
based renderers so some processes can be reused among the views. The method they
propose can also be used for creating other architectures when it is integrated with
different architectural patterns. For instance, if the parallel rendering is used with the
remote rendering, the setup will describe a kind of architectures falling into the 1-n
Flavor, which is another Flavor of the 10n Graph Style. Similar to 1-1 architectures, 1-n
architectures also distribute simulation and presentation to different nodes. But the
difference is that multiplayer games with player-specific views are supported through
multiple PWs. In [121], Bangun and Beadle propose an architecture splitting the
simulation and rendering processes of multiplayer games. The approach uses CPU
servers to simulate the game state, and game consoles (e.g. Sony PlayStation, Nintendo
64) to render the game scenes and capture user input. The CPU server generates 3D/2D
data related to the game state, and sends it to the game consoles. The architecture uses
the simulation capability of CPU servers and the 3D/2D processing capability of game
consoles and thus achieves an efficient resource usage. Streaming 3D/2D data also
decouples game clients from many modules of game engine, which makes it easy to use
heterogeneous devices.
1-n architectures use a single GW to maintain a consistent Game World, and multiple PWs to support more than one player in the game. Advantages and challenges of 1-n architectures are similar to above-mentioned 1-1 architectures. Support of multiplayer games with player-specific views is the major improvement compared to the 1-1 architectures.

6.3.3 Graph Style Inn Architectures
The Inn Graph Style features a single GW connected to multiple LWs and PWs. Our review shows a great diversity of architectures falling into this category, resulting in 4 Flavors/sub-categories. All these Flavors use a remote Connector Type to connect the GW and the LWs, meaning that they all represent network game architectures. The
analysis of the architectures revealed the following three differences between the Flavors:

1) The LWs and the related PWs could be simulated on the same node or not. ($1-nn$ vs. $1-nn$)
2) The LWs could be inter-connected or not. ($1-nn$ vs. $1-n(\cdot)n$)
3) The GW and the related PWs could be connected or not. ($1-nn$ vs. $1-nn$)

Among the four Flavors, $1-nn$ is found to be the most popular one, and other Flavors can be seen as extensions of the $1-nn$ Flavor in different ways.

6.3.3.1 Flavor 1-nn Architectures

The $1-nn$ Flavor denotes that the LWs and the PWs are natively connected, meaning that they are simulated on the same node. Since the PW is the Game World that a player directly interacts with, the node is usually the computer of the player. The single GW is remotely connected with the LWs, meaning that the single GW is simulated on a node other than the players’ machine, which is usually a dedicated server. In the reviewed articles, three architectures fall into the $1-nn$ category. By analyzing the GWG Flavor, the following attributes of the architectures can be identified:

1) The player’s machine maintains the local game state, rendering the game scene locally. The architectures with local rendering use the player’s resources for rendering computation thus lessen the server capability requirements.
2) The GW is maintained on a dedicated server, which the players do not have direct access to. This makes such architectures better protected against cheating.
3) The single GW implies the risk of single point of failure.

One example of the $1-nn$ Flavor is an architecture proposed by Caltagirone et al. for a massively multiplayer online role playing game engine [97]. Typical Client-Server pattern is used in the engine architecture and the layered system pattern is also used. The authors introduce the basic architecture with a single server and they also are aware of the scalability issues connected to a single central server, so they also discuss a possible extension of adding more servers to the basic architecture. Another example is Strifeshadow Fantasy (SSF) [107], a massive multiplayer online role-playing game with over 10,000 registered players. In SSF, a central server hosts the SSF server program simulating the entire Game World. Players connect to the server through the SSF client applications. HTTP is used for client-server interaction, and Macromedia Flash 4.0 is used to render the game scene. Clients regularly send Update Request Messages (URM) to the server, which is based on Active Server Page (ASP) combining the use of HTML, scripts, and Microsoft ActiveX server components. The server also uses a relational database for storing and retrieving the game state. Similar to SSF, SharedWeb [123] is an architecture for a 3D War Game over WWW. HTTP is also used in SharedWeb as the primary communication protocol between the client and the server, and since it is designed for 3D War Game and war simulation, 3D rendering is necessary. SharedWeb uses a specific web browser instead of Macromedia Flash on client to do 3D rendering, and the web browser supports and coordinates 3D rendering, Internet Relay Protocol based chatting, and HTML documents.
By examining the three architectures falling into 1-\textit{nn} Flavor, we can further summarize the characteristics of the architectures. 1-\textit{nn} is a straightforward and easy-to-implement solution for networked multiplayer games. The single GW in 1-\textit{nn} architectures is usually simulated on a server with high computational power and network capability, which enables multiple players to share a relatively complex Game World. Player resources are used to do rendering, which releasing the server from this resource-consuming task. The centralized server makes it easier to do cheating-prevention and charging for services. Single point of failure is a major disadvantage of the 1-\textit{nn} architectures. The single server bottleneck also prevents 1-\textit{nn} architectures from being suitable for modern MMOGs with very complex scenes and a large amount of players. However, the 1-\textit{nn} architectures is one step in the right direction for supporting MMOGs.

\subsection*{6.3.3.2 Flavor 1-n-n Architectures}

The 1-\textit{n-n} Flavor has also a single GW, but unlike 1-\textit{nn} architectures, the LWs and PWs are remotely connected. This means the presentation and management of related local states are distributed to two nodes.

In [98], Quax et al. present a Remote Rendering Server architecture for Networked Virtual Environments (NVEs), which can be the bases for computer games, but can also be used for other purposes. Quax et al. use rendering servers that are in charge of maintaining the local state, rendering and encoding rendering result to video streams which are played by thin clients such as mobile phones. A central world server is used to simulate the GW as in the other 1\textit{nn} architectures.

\subsection*{6.3.3.3 Flavor 1-n(-)n Architectures}

The difference between the 1-\textit{n(-)n} and 1-\textit{nn} Flavor is that the former adds a remote Connector Type between LWs. The connectors can be used for exchanging local states, which reduces client latency and lessens the requirement of server bandwidth. This is because the direct client-to-client communication removes the need for all communication to go through the server.

The solution proposed in [125] is an example of the 1-\textit{n(-)n} architectures. Events sent by players are divided into two kinds: state-changing event (e.g. pick up a coin) and non-state-changing event (e.g. player movement). The state-changing events are sent to a central server owned by the game operator, and the non-state-changing events are sent to a “regional server”, which is a player’s machine in charge of forwarding events to players in that region. Note the difference between this “regional server” and the regional server in Multi-Server Partitioning architectures: for the latter, the regional servers simulate GWs. With regional servers, the architecture utilizes network bandwidth of the players, and sends non-state-changing events only to those relevant in the region to save the overall network load. The architecture improves the scalability and performance quality attributes of typical Client-Server architectures, but whether an event is a “state-changing event” or not, might be highly dependent on the actual game design and is therefore hard to determine.
Another example of the $1-n(-)n$ architectural pattern is the approach proposed by Pellegrino and Dovrolis in [99]. This is a hybrid architecture named Peer-to-Peer with Central Arbiter (PP-CA), which modifies Peer-to-Peer architecture by adding a central arbiter, resulting in a simple consistency resolution and scalability at the same time. Games based on this architecture run on player’s client devices as well as on a central server (arbiter). Player clients exchange updates by direct communication. Whenever a player client sends an update, it also sends an update to the central arbiter, and the arbiter will check for state inconsistencies, and correct the player clients if inconsistency is detected in their local state.

### 6.3.3.4 Flavor 1-nn- Architectures

The remote Connector Type between the GW and PWs is the main characteristic of $1$-nn- Flavor, meaning that some of the PWs are generated on the same node where the GW is simulated, while others are generated on the nodes simulating the LWs. Architectures that belong to this GWG Flavor support both client and server rendering.

Only one of the architectures from our review falls into $1$-nn- Flavor, which is the Pervasive Multiplayer Multiplatform Game (PM2G) architecture [124]. In PM2G, players are mostly connected to the virtual world, but they may also play against each other using heterogeneous devices. To enable PM2G, Trinta et al. propose a PM2G architecture employing the client-server pattern as well as the layered system pattern. The PM2G architecture includes three layers: the PM2G server, the PM2G services, and the PM2G client application. The architecture supports user-context management and content/interaction adaptation through the PM2G services layer. Context includes relevant information about the player’s connection to the virtual world, such as his or her device and preferences. The context is used by the PM2G architecture to provide the content adaptation service, which is a process to translate game state to a proper data representation according to the player context. Players using thin clients may receive the text-based game state, while players using more powerful clients may receive the default game state and their devices will do the rendering and presentation. The text-based game state is actually the rendering result, which is produced by the server through translating and serializing of a part of the GW that is within the area of interest of a specific player. Streaming text-based game state is similar to streaming 3D data or video in the computation distribution aspect, but requires much lower bandwidth and simpler player clients.

Supporting both client and remote rendering makes the $1$-nn- architectures more flexible by enabling more heterogeneous player clients compared to $1-nn$ architectures. However, the approach that only supports text-based remote rendering has limited usefulness. An approach that provides both local and remote graphical rendering can be a topic for further research.

### 6.3.4 Graph Style $n\emptyset n$ Architectures

The $n\emptyset n$ Graph Style features multiple GWs and PWs, where the PWs are directly connected to GWs without LWs. This means that the game externalization does not rely on a local state. Possible Flavors of the $n\emptyset n$ Graph Style include the $n-n$, the $n(-)n$, and
the \(n\)-\(n\). However, only the \(n\)-\(n\) Flavor was found in the articles included in our review.

The \(n\)-\(n\) Flavor features a native Connector Type between the GWs and the PWs, and a remote Connector Type between the GWs. The PW is the Game World players directly interact with, so it is always simulated on the player’s machine. Considering that the GW and the PW are natively connected, the GW is simulated on the same player’s machine. Since all Game Worlds are simulated on players’ machines, we name such architectures Peer-to-Peer architectures. Note however, that the \(n\)-\(n\) Flavor is not the same as the Peer-to-Peer architectural pattern, because the GWG framework classifies game architectures from the Game World distribution perspective, which is different from the network connection perspective. In traditional taxonomies, when the server is running on some players’ machines, the architecture will be also called “Peer-to-Peer architecture” from the network connection’s perspective in some articles and “public server architecture” from the responsibility distribution’s perspective in some other articles. With the GWG framework, this kind of architecture is classified into the \emph{nnn} Graph Style but not the \(n\)-\(n\) Flavor that we are discussing here, because player machines that run servers are considered as GW hosts and player machines running pure clients are considered as LW and PW hosts.

By analyzing the characteristics of this Flavor, we can find the following advantages and challenges of \(n\)-\(n\) architectures.

\textbf{Advantages:}

1) Using player resources to simulate all Game Worlds can provide better scalability;
2) Multiple GWs are simulated, and it can potentially avoid single point of failure;

\textbf{Challenges:}

1) Because the GWs are simulated on player machines and are inter-connected, maintaining state consistency and persistency becomes difficult;
2) Players can directly access the GW, which brings challenges such as anti-cheating.
3) For some MMOGs with complex game worlds, simulation of the whole GW demands a lot of resources from the player machines, making it a challenge to implement \emph{interest management} that limits the objects account on a node.
4) Each player machine is responsible for sending updates of the GW it is simulating to other clients. This causes corresponding network traffic, which requires optimization.

The works described in the reviewed articles are mainly focused on dealing with above challenges.

Due to the scalability issues of network level multicast protocols, Application Level Multicast (ALM) is practically adopted in some architectures to optimize network
traffic. A peer-to-peer game architecture with a tree-based multicast approach is proposed in [126], where player nodes are connected to each other forming a multicast tree. The nodes that are found to minimize latency from all players will be selected as the “center” of the tree. We also found another example of tree based ALM in [127]. Interest management is usually managed by ALM due to the requirement to limit network traffic. Rooney et al. introduce a peer-to-peer architecture with ALM in [128], which use interest management to organize nodes according to their area of interest. Each area of interest may be connected with a “multicast reflector”, which is responsible for forwarding messages sent by nodes in the area of interest to all subscribers.

To maintain a consistent global state among nodes, various protocols are proposed. One solution is VoroGame [108]. A structured P2P overlay based on a distributed hash table (DHT) (e.g. [109, 110]) combined with an unstructured P2P overlay based on a Voronoi diagram [111] is used to manage the global state. The Game World is divided into zones based on a Voronoi diagram, and DHT is used to store game objects on peers evenly and fairly. Each peer has two roles in the architecture: 1) Voronoi responsible related to a zone in the Voronoi diagram, which is in charge of simulate behavior of objects inside the zone that the peer locates in, and 2) DHT responsible that is in charge of storing and forwarding object state changes from Voronoi responsible to players that are interested in the objects state. The use of a “super node” is a common solution to maintain state consistency, and the super node can solve the inconsistency by providing the final decision of the global state or part of the global state. An example is a hybrid architecture that is proposed in [127], where the Game World is divided into hexagonal zones. Each zone has a super node in charge of coordinating the interaction among players in that region. Super nodes can also be used to detect cheating because a super node could recognize a manipulated game state. An important benefit of the super node approach is saving service cost by using player resources. Mediator [129] is a p2p architecture using resources provided voluntarily by the player. Four kinds of super nodes (mediators) are introduced and a player can announce their resource (computation power, bandwidth, etc.) and be assigned tasks by the super nodes. They can also be promoted to super nodes when needed. A reward scheme according to player contribution is also discussed in [129].

The super node can be a security threat since the system does not have complete control of the node and the super node and player client can be the same machine. ACORN [130] is a Peer to Peer architecture with dynamic coordinator (super node). The coordinator could be moved from one node to another in order to make cheating hard if not impossible. Coordinator Access Point (CAP) is introduced to help peers to find the coordinator. In DACA [131], super nodes of regions are selected from the player nodes whose avatars are not in the corresponding regions. In [132], coordinators (super nodes) which are in charge of objects states in a region are randomly selected. The objects can be divided into subsets based on types, and the objects of each type are managed by a coordinator. The coordinator is also the region server for a region. Interest management and multicast are also discussed in [132]. Players in each region form an interest group, and transient state updates of a player will be multi-casted to the group, and if the game mechanics allows, to the players in another group who have subscribed to these state
changes. Besides the state consistency, state persistency is also a research direction. Hampel et al. present a Peer-to-Peer MMOG architecture in [103], where Pastry is used to implement the P2P network infrastructure, and its extensions, Past and Scribe, are used to implement persistent object storage and event distribution.

The literature review reveals two challenges of \( n(-)n \) architectures apart from those addressed in the GWG reasoning:

1) It is difficult to charge players for gaming services when players do not connect to any centralized server managed by the operators.
2) A reward mechanism must be applied to encourage players to contribute with their computational and network resources.

It is possible to foresee several more \( n0n \) graph Flavors, however, all architectures in our review fall into the \( n(-)n \) Flavor. This may indicate a potential research direction of exploring un-implemented \( n0n \) Flavors. It is also worth mentioning that \( n0n \) architectures are mainly proposed by researchers and are rarely (if not never) used in commercial games.

### 6.3.5 Graph Style \( nnn \) Architectures

\( nnn \) architectures attract most interests from researchers. Multiple GWs means that the global state is distributed or duplicated, while multiple LWs with PWs means that multiple instances of local state are maintained with multiple player views. We found 4 different Flavors in literature, which are \( n-nn, n(-)-nn, n-n(-)n, \) and \( n(-)-n(-)n \). The major differences between these Flavors are:

1) The GWs are connected with each other or not: \( n-nn \) and \( n(-)-n(-)n \) vs. \( n(-)-nn \) and \( n(-)-n(-)n \)
2) The LWs are connected with each other or not: \( n-nn \) and \( n(-)-nn \) vs. \( n-n(-)n \) and \( n(-)-n(-)n \)

#### 6.3.5.1 Flavor \( n-nn \) Architectures

When multiple GWs are not interconnected but remotely connected to LWs, \( nnn \) architectures fall into the \( n-nn \) Flavor. The disconnected nature of the GWs normally means they do not share a common state, so there is no need for synchronizing the GWs. Thus, the entire global game state is partitioned. Partitioning the Game World can reduce computation load and network traffic demands on one single server, and thus make the architecture more scalable.

An intuitive partitioning approach is spatial partitioning, i.e., dividing a single Game World into zones based on virtual spatial distribution. I name such architectures Zone-Based Architectures. One example is presented in [133], where the virtual world are divided into regions. Each region is a disjoint piece of the GW and it can be any convex polygon. Each region is assigned to a server that is responsible for simulating this piece of the GW. Each player client will only exchange messages to a server simulating the piece of Game World holding his/her avatar. The spatial partitioning could also be dynamic instead of predefined. Real Tournament (RT) [134] shown in Figure 6.5 is an
augmented reality FPS game providing a multiplayer mode. Each player joining the game will be assigned to a zone of the Game World, and will then be in charge of the objects’ states of that zone. The spatial partitioning is dynamic, meaning that at the beginning, the first player is in charge of the whole world, and when a second player joins, the whole world will be divided into two zones and assigned to two players respectively. As more players join, the zones will be further sub-divided. This architecture utilizes the characteristics of location-aware games that the proximity of avatars implies proximity of players in the real playgrounds, and thus achieves both the network optimization and the computation load balance. Another dynamic partitioning architecture is proposed in [135]. The Game World is divided vertically against the x-axis in the architecture, and a server node manages each partition. If a node is overloaded, it can assign some game units (object or characters) to its neighbor by resizing the partition it is processing. Both of the approaches keep the spatial continuity of the partition on each server, when the system dynamically adjusts the partitions. But it is not always necessary. The Game World could be further divided into microcells [27], and a group of microcells forms a zone dynamically. The microcells belonging to the same zone do not have to be adjacent, thus non-spatially-continuous partitioning is supported. The microcell concept is proposed by Vleeschauwer et al. in [27] for an architecture that balances processing load among multiple servers. One server manages one or many microcells to achieve a balanced server allocation. A similar approach is introduced in [137], where Hori et al. divide game space into grids and the grids are dynamically assigned to regions. An alternative approach to spatial partitioning is object-based partitioning. This approach detaches the relationship between virtual location and game objects, and assigns game objects to servers based on other object attributes. Hsu et al. propose a Clustered-Server architecture in [93], where a single Game World can be distributed to a group of connected servers. Their work does not focus on a specific partitioning approach, but presents some possibilities of the world allocation, e.g. region oriented, and (virtual objects) hierarchically structured.
Another research topic of the \textit{n-nn} Flavor is how to “efficiently” assign partitions to servers. The classical solution is to assign partitions according to server load to optimize utilization of server capability. However, such approaches disregard the player context in the real world. E.g., a player may be assigned to a server that is far away from his physical location. An interesting solution is proposed in [138], which describes an architecture with geographical distributed proxies (zone servers) to minimize the latency due to geographical distance. The proxy to be selected for a zone is decided by the overall geographical proximity of the players in that zone and the proxies. Another efficient way to assign zones to servers is based on players’ activities in the zone. Distributed Entertainment Environment (DEE) [139] is an online game architecture developed at BT laboratories. DEE divides the Game World into predefined partitions, and a process will manage each partition. If the state of a zone becomes completely predictable over time, its state is written back into a persistent object store, and the relevant process is closed. If a new activity occurs within a zone, a new process is started and the previous zone state is restored.

Most of the above solutions use dedicated servers controlled by a game service provider. However, an attractive alternative can be to use players’ computer resources to reduce service costs (public servers). In [140], the authors discuss the dynamic partitioning of a Game World as the player number increases, and the usage of public servers to simulate a zone in the game space. Similarly, a public server architecture is introduced in [141]. In this architecture, some player nodes will be used as Regional Controllers (RC). The GW is divided into regions, which are simulated on RCs. Instead of the conventional approach of putting one server in charge of one or more regions, this architecture uses a group of RCs to simulate one region. The game state of the region on each RC in the group should be the same. The RCs do not communicate with each other to keep consistent game state as the consistency depends on a voting system running on the clients, which should also keep the system cheat-proof.

The \textit{n-nn} architectures use multiple instances of GWs to distribute simulation to multiple nodes. The multiple isolated GWs reduce the computational load and reduce network traffic on a single server without introducing consistency problems. The major advantages of \textit{n-nn} architectures are the better scalability than \textit{1nn} architectures, and simpler consistency control than \textit{n(-)-nn} architectures (we will discuss this issue later in this chapter). The major challenge of \textit{n-nn} architectures is that as they simulate the Game World on different nodes, the players may have to connect to a node that is not network-wise close to the player.

\textbf{6.3.5.2 Flavor n(-)-nn Architectures}

The remote Connector Type between multiple GWs differentiates \textit{n(-)-nn} from \textit{n-nn} architectures. The interconnections between GWs require them to be synchronized, which implies that the GWs are either identical or sharing common parts.

We name architectures with identical interconnected GWs as Mirrored Multi-Server Architectures. Rokkatan [142] is a multiplayer online game based on this kind of
architectures, which is shown in figure 6.6. The game state is simulated on multiple servers, and each server manages a full instance of the entire game state. The servers use a cooperation protocol to keep the game state consistent. A player joining the game can select the fastest server to get the smoothest gaming experience.

![Figure 6.6 Rokkatan][142]

Another example is the Tree-Based Server-Middlemen-Client (TB-SMC) architecture proposed by Matuszek [151]. A description of how the TB-SMC architecture is evaluated in terms of performance by developing a simulator can be found in [143]. TB-SMC architecture uses two servers that simulate the same Game World. One is the primary server in charge of updating all clients, and the other is the backup server that is used to provide continuous service when primary server fails. The middlemen are nodes that are used to forward messages from the server to the clients and to send messages collected from the clients to the server. The nodes are connected in a tree, and the number of middlemen as well as the depth of the tree can be adjusted when number of players increases or decreases. A double central server used in TB-SMC architecture also reduces the risk of single point of failure. Hydra [144] proposed by Chan et al. is a similar solution using a double central server, but in this approach the player machines are also used to simulate GWs. Each player node acts as both client and server. The Game World is divided into zones and each zone is called a “slice”, Hydra replicates slices with multiple copies, and each node can be in charge of one or more slices. Each zone has a primary slice simulated by a server, which keeps the state up to date according to player client messages. The backup slices servers are responsible for updating backup slices. Another kind of $n(-)-nn$ architectures features the combination
of the GW duplication and the GW partition. One example is the public server architecture proposed in [145], where a central server is used to process log-in and manage sub-servers. The sub-servers are selected from player nodes. Each sub-server is assigned with a partition of the GW and thus becomes the partition responsible. Each sub-server also maintains a copy of each adjacent partition managed by other sub-servers. A partition is simulated by its partition responsible, and its copies maintained on other sub-servers are updated accordingly through network communication. When a server fails (e.g. a player leaving the game), the central server could recover partition state using copies maintained by other sub-servers, and the central server will take charge of the partition for a period until a new sub-server is selected. The partitions can also be sub-divided and area servers are introduced to manage the sub partitions, which are also player clients.

In n(−)-nn architectures with non-identical GWs, servers simulating different GWs may have different purpose or authority. E.g. a Generic proxy architecture is proposed in [102]. A central server and multiple proxies are used to coordinate the game state updates. The server has the final authority, and the proxies are trusted to some extent because players do not control them. Proxies act as an extension of the central server and players can choose to interact through the proxies. Another proxy architecture is proposed in [146], which uses geographically distributed proxies to handle “time critical” computations like simulation of player movement. Here, the players connect to a nearby proxy, and send input events to the proxy. The proxy will simulate the players’ movement, send a new state to the players, and multicast the state to other proxies if necessary. The global game state except the time critical state information will still be handled by a central server, or by multiple servers if the world is divided into zones.

The advantages of n(−)-nn architectures are geographical optimization of network communication as well as avoiding a single point of failure. The major challenge of n(−)-nn architectures is how to develop a protocol that insures consistency.

6.3.5.3 Flavor n-n(−)n Architectures

The n-n(−)n Flavor represents an extension to the n-nm architectures by adding a remote Connector Type between LWs. This means that the local states of the players can be exchanged. This approach is usually used to improve performance. By enabling direct player-to-player interaction, client latency will be reduced compared to traditional communication protocols that have to go through a server. Due to the direct connection between clients, the n-n(−)n architectures are often named Peer-to-Peer architectures in traditional taxonomies. However, the GWG framework has shown that the Peer-to-Peer architectural pattern can represent quite different approaches, which are represented in multiple Graph Styles and Flavors. So we name the n-n(−)n architectures Multi-Server Partitioning with Client Communication.

ALVIC-NG [147] is a proxy-based architecture, which uses central servers to simulate the game state, and a group of proxies to reduce communication load on the servers as well as on the clients. The proxies act as mediators for server-client communication and are not in charge of the computation of the game state. The game state on the proxies can also serve as cached data in the system to reduce latency in some cases, so it can be
mapped to the LWs in the GWG framework. The architecture divides the world into regions managed by logical servers. The regions can be split or reassigned to servers that can dynamically be added. Similarly, Mirrored Arbiter (MA) [115] is an architecture using multiple arbiters (servers). The Game World is partitioned into regions. Each arbiter is responsible for state computation and ensures consistency of a region. Clients are fully connected and communicate directly to exchange states. Clients are also kept connected to the arbiter in order to deal with state consistency. HERA [148] is a public server architecture which groups peers into servers and clients. The servers are in charge of managing the game objects. Clients within the same Area Of Interest (AOI) are organized as a fully connected graph in order to directly communicate with each other to increase interactivity.

The n-n architectures enable the exchange of the local game state without going through the servers and thus decrease latency. However, a problem related to such architectures is security, i.e. the game state exchanged must not be misused in order to break the balance or gain improper advantages over others players.

6.3.5.4 Flavor n(-)-n(-)n Architectures

Intuitively, the n(-)-n(-)n architectures extend n(-)-nn architectures through allowing direct communication between player clients to reduce network latency and network load. The Enhanced Mirrored Server (EMS) [149] architecture is one example of this Flavor. The peers (clients) in EMS can communicate directly to exchange game state updates, and multiple mirrored servers are used to reduce latency. The connection between clients reduces network latency and also reduces network and computation load on the server. Like the n(-)-nn Flavor, the benefit of simulating the entire game state using multiple servers comes with the challenge of synchronizing the game state.

The nnn architectures are leading-edge techniques that attract the most of interests from researchers in this field. Multiple LWs and PWs implies multiplayer mode, and multiple GWs can potentially reduce players as well as objects density on a single node thus reduce computation and network load. The key characteristic of nnn architectures is the use of multiple GWs. We have identified three main challenges related to such architectures:

1) It is difficult to keep consistency among GW instances when they are interconnected;
2) Single point of failure when GWs are isolated; and
3) Difficult to implement.

6.4 The Roadmap of Game Architectures Evolution

Game architectures have kept evolving ever since the first computer game was developed. The roadmap started with II architectures providing a single game software with a single view on the same machine. Although it is relatively simple, it is still the most appropriate solution for most single player commercial games today. An extension is multiplayer game based on the II architectures where players share a single view, which is natural and convenient for some game genres (e.g. 2D action games and shooting games), but inappropriate for games that require individual player views (e.g.
racing games). To support the latter, the *In* architectures must be used which supports multiple player views (PWs) rendered by a single program. The *I* and *In* architectures have dominated the game industry for ages before the *I-nn* architectures opened the door for online games. The *I-nn* architectures provide sharable and consistent Game Worlds, centralized control, and less chances of cheating. However, the *I-nn* architectures are limited in terms of scalability, which prevents them from being used in MMOGs.

Work addressing the scalability issue of the *I-nn* architectures has been done in two directions: 1) use more servers to share computation load, and 2) use player resources to reduce server load. The first approach is represented by the *n-nn* architectures, where the GW is divided into predefined pieces (e.g. regions), and players in different places of the Game World may be assigned to different servers. Partitioning of the Game World is the major characteristic of this kind of architectures, thus named Multi-Server Partitioning architectures. Spatial partitioning is the most popular solution to divide the game world, but object-based partition has also been explored. World partitioning can further be extended to dynamic partitioning, where strategies such as resizing and Microcell are applied to achieve better load balancing.

The second approach is represented by the *n(-)n* and *n-nn* architectures. The former is named Peer-to-Peer architectures where the GW is simulated on players’ machines, and where the main challenge is to keep the global game state consistent. The latter is named Public Server architecture where selected player machines serve as the hosts for simulating the GW, while clients connect to the hosts just as they do in the Multiple-Server Partitioning architectures. The possible problems related to the use of player resources are issues related to security and availability, which motivate research on such architectures.

Both *I-nn* and *n-nn* architectures may cause high latency to some players in MMOGs, since players are distributed over the world and the server they are connected to may not be geographically or network-wise close. A solution to this issue is the use of proxies (mirrored servers), which host the same GW but are distributed to different geographical regions. This solution is implemented in *n(-)-nn* architectures, where the remote Connector Type between GWs implies that the state synchronization is carried out among mirrored servers. Besides the latency issue, single point of failure is also eliminated when using *n(-)-nn* architectures.

Above-mentioned architectures represent the trunk of the game architecture roadmap, and there are also some branches connected to two major architectural strategies. One strategy is Remote Rendering. When thin-clients such as set-top boxes and low-end cellphones which are not capable of real-time graphical rendering are used in a graphical game, server rendering is necessary. The graphics or image based rendering results can then be streamed to the thin-clients over the network. The strategy can be combined with many GWG Flavors: when Remote Rendering is used in combination with the *I* architectures, it will result in the *I-I* architectures. When Remote Rendering is used in combination with the *I-nn* architectures, it will result in the *I-n-n* architectures. This strategy can be open for further exploration, as potential
architectures such as $n$-$n$-$n$ have not yet been proposed. Another strategy is Client Communication. When the GW and the LW are distributed to two kinds of computational nodes, the synchronization of the LWs is usually done through a LW-GW-LW route, which brings additional latency. Allowing direct state exchange among LWs can solve this problem. Combing the Client Communication strategy with various GWG Flavors results in corresponding extensions: when Client Communication is combined with the $l$-$nn$ architectures it will result in $l$-$n$($-$)nn architectures, and when Client Communication is combined with the $n$-$nn$ and $n$($-$)-$nn$ architectures, it will result in $n$-$n$($-$)n and $n$($-$)-$n$($-$)n architectures.

Architectures are developed for fulfilling specific functional and non-functional requirements. So it makes no sense to say which architecture is the “best” in general, but for a given problem domain, there may be one or more architectures which are more appropriate than others. Table 6.3 shows which game types each GWG Flavor is usually applied to, i.e. for a game type which architecture sub-category(s) is appropriate in practice.

<table>
<thead>
<tr>
<th>GWG Flavor</th>
<th>Game Type</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$-$l$</td>
<td>Single-Player Game on Thin Client</td>
<td></td>
</tr>
<tr>
<td>$l$-$l$</td>
<td>Typical Single-Player Game, Multiplayer Single-Screen Game</td>
<td></td>
</tr>
<tr>
<td>$l$-$n$</td>
<td>Multiplayer Online Game on Thin Client</td>
<td></td>
</tr>
<tr>
<td>$l$-$n$</td>
<td>Multiplayer Split-Screen Game</td>
<td></td>
</tr>
<tr>
<td>$l$-$nn$</td>
<td>Multiplayer Online Game</td>
<td></td>
</tr>
<tr>
<td>$l$-$n$-$n$</td>
<td>Multiplayer Online Game on Thin Client</td>
<td></td>
</tr>
<tr>
<td>$l$-$n$($-$)n</td>
<td>Multiplayer Online Game with Thin Client and Standard Client</td>
<td></td>
</tr>
<tr>
<td>$l$-$n$($-$)n</td>
<td>Multiplayer Online Game</td>
<td></td>
</tr>
<tr>
<td>n($-$)n</td>
<td>Multiplayer Online Game, MMOG</td>
<td></td>
</tr>
<tr>
<td>n$-$nn</td>
<td>MMOG</td>
<td></td>
</tr>
<tr>
<td>n$-$n$-$nn$</td>
<td>MMOG</td>
<td></td>
</tr>
<tr>
<td>n$-$n$($-$)n</td>
<td>MMOG</td>
<td></td>
</tr>
</tbody>
</table>

As we showed in [48], the GWG framework can also be used for exploring future architectures. The basic idea is that if we create a list of all possible Graph Styles and Flavors, and fill the list with the data gained from the systematic review, we can find some “holes” (unused Graph Styles and Flavors) in the list. Each unused Graph Style or Flavor may imply an unknown game architectural pattern, which can serve as a base for exploring unknown architectures. Further, we have presented an example of creating a Hybrid Remote Rendering Architecture with the $1nn$ Graph Style in [48] to validate the method. The architecture is appropriate for mobile and ubiquitous games, and it has not been found in any existing article in the systematic review. Fitting existing architectures...
into the GWG framework is a necessary step to exploring new architectures with the framework, so it is also an important contribution of this chapter.

6.5 Summary

The game architectures are getting more and more complicated, and thus attracting increasing interests both from researchers and practitioners. The systematic review summarizes studies and findings on game architectures in the literature. Through the review, the usefulness of the GWG framework as an architectural taxonomy was validated.

From Single Software to Remote Rendering architectures, from Multi-Server to Peer-to-Peer architectures, this chapter showed a big picture of game architecture research and technologies. Game architectures that do not include network are rarely described in literature, although these are the popular architectures in commercial single player games or multiplayer games running on one machine (a PC or a console). Online game architectures, especially MMOG architectures attract most of the interests from the researchers. Multi-server architectures (\textit{nnm Graph Style}) and Peer-to-Peer architectures (\textit{n0n Graph Style}) represent the two major architectural patterns for MMOG in game architecture research. For Multi-server architectures, partitioning of the global Game World and state consistency are issues being widely discussed. For Peer-to-Peer architectures, more design issues are addressed by the studies in literature (see [118] for more details).

The architectures reviewed may feature one or more architectural strategies, e.g., zoning, use of player resources, etc. These tactics are constructed to handle particular architectural drivers such as reducing server computational load and anti-cheating. This chapter has described and discussed these tactics and drivers. It will be further synthesized in future work. The systematic review only includes architectures described in the literature, and there are also a lot of architectures of popular commercial games without any public available documentation. Applying the GWG framework in analyzing these architectures may produce more interesting results. However, it is difficult to gain the access to such architectures, especially for games that are not open source. This limitation is worth noting and it is a future work of my Ph.D. study.

In next Chapter, I will discuss the application of GWG on supporting MDGD. The review results presented in this chapter is an important input to the next step of research that is about MDGD addressing game architectural diversity. Thoroughly investigating the game architecture domain and gathering information about the characteristics of the modern architectures is essential for making software architectural decisions in MDGD, and I believe this chapter can not only contribute to my Ph.D. study but also help other MDGD developers when they need to work with various game architectures.
7 Addressing Architectural Concerns In MDGD Using GWG

An important application of the GWG framework is to help addressing architectural concerns in MDGD. In this chapter I will discuss an approach that integrates the concepts from the GWG with the DSL to include architectural information in game models. With this architectural information, the code generator is able to generate the code for the target architecture. The basic idea of the approach is to use the Game World concepts to organize model elements at a high level, and the Connector concepts to specify the architectural details at a low level. The split of two levels of concerns improves the usability on one hand, and loosens the dependency of the model to the generated architecture on the other.

7.1 The Need For Supporting More Architectures

A significant drawback of current MDGD approaches is the lack of support for multiplayer architectures. All of them presume a standalone software architecture, which provides very limited opportunities for multi-player gameplay: typically only multi-player on shared-screen is possible. As it was shown in Figure 7.1, the local multi-player gameplay is still used in modern games such as Wii Sports (Nintendo 2006) Super Street Fighter IV (Capcom 2010) and Mario Kart 8 (Nintendo 2014). However if we consider the most successful commercial games, most of them support internet-based interactions.
The Internet provides more opportunities for player-to-player interactions and extends the participation of a game to players across the world. The Internet enables several game modes, e.g. host-joiner mode such as in StarCraft II: Wings of Liberty (Blizzard Entertainment, 2010), Massively-Multiplayer Online (MMO) mode such as in World of Warcraft (Blizzard Entertainment, 2004), and Multiplayer online battle arena mode such as in League of Legends (Riot Games, 2009). Figure 7.2 shows the screenshots from the above mentioned Internet-based multiplayer games.
To support a diversity of internet-based games, standalone software architectures are not sufficient. As a result a variety of online game architectures have been developed, which I have discussed in Chapter 6. MDGD as a game development methodology has to support online architectures as well in order to be useful in practice. However, none of the existing MDGD approaches reviewed in Chapter 2 has addressed these architectural concerns, making it impossible to validate their usefulness with regards to complex architectures other than standalone software architectures. This means that there is a technical gap between the academic research of MDGD and industrial game development. In this chapter I will discuss the problem of supporting a diversity of architectures in MDGD, and propose a high-level solution based on the GWG framework presented in Chapter 5. The use of the GWG in MDGD can raise the abstraction level of game architectural concerns, which makes the game models independent of the concrete architectures on one hand, and releases the modelers (game designers) from working with the architectural details on the other. In Section 7.3, I will describe how the concepts from the GWG framework are used to grant game DSLs the capability of specifying architectural information in game models. The information is used by a corresponding code generator that supports the target architectures to generate architecture-specific infrastructure code, which is executable on a particular
domain framework. This method will be further evaluated through a case study in Chapter 8.

7.2 Include Architectural Information In Models: The Problem

In Model-Driven Development, if the generated artifacts are architecture-specific, the model must contain all relevant information for the generation of the infrastructure code to support the target architecture [60]. This is also true in MDGD, and I will illustrate this fact with an example modeling the game Orc’s Gold. The original game of Orc’s Gold was described in Chapter 4, and now we assume that the game needs to be extended to a networked multiplayer version. The major characteristics of the extended game are:

1) The game can be played by two players at the same time on their own machines respectively. Each player controls one main character, and they play cooperatively to achieve the shared goal: collecting all the chests in the map.

2) The game state is maintained by a central server, which sends the update of the game state to player machines periodically. The player machine is in charge of handling player input to generate the movement command, sending the command to the server, and maintaining a local state for presentation as well.

3) The game still has four AI patterns (Guard, Dragon, Chest and Tree). The Guard, Dragon and Chest are considered as the global AI behavior, which influences all players participating in the game, but the Tree is considered as the local AI behavior which has nothing to do with the global gameplay, and only affects the local presentation. Therefore to save resources on the server, the Tree pattern can be deployed directly on the clients.

Since the new game has adopted client-server architecture, the generated code should be executed by a framework supporting this architecture pattern, and the code itself has to be divided into a client part and a server part as shown to the right in Figure 7.3.

![Figure 7.3 RAIL Model and The Desired C-S Code](image-url)
The Guard and the Dragon Pattern are generated as part of the server code, because the game state they manipulate must be shared by all the players. To synchronize the state between the clients, a proxy is created for each Pattern on the client to receive and translate the state updates. The Flower Pattern does not affect the game behavior, and is created only for improving the visual effect of the game. It is not necessary to synchronize the state it manipulates among the players, so it can be deployed only on the client.

In Chapter 4, we generated the single-player version of Orc’s Gold from the model as shown to the left in Figure 7.3. However, it is not possible to generate the game software to the right in Figure 7.3 from the same model, because the information about which pattern needs to be synchronized is not contained in the model. Without such information, the code generator is not able to deploy the generated code to the right part of the software and in addition create corresponding proxies. To solve this problem, architectural information must be contained in the model, which requires the language level support from the DSL. A straightforward solution is that we add a property named "deploy_to" to the Pattern construct in RAIL, with the possible value of "server" and "client". The modeler can then specify where the pattern should be deployed to by modifying the value of the property. The architectural information will be taken as the input to the code generation, and the software on target software can thus be generated as expected. This solution has two significant drawbacks:

1) The modeler must have the knowledge about the software architecture. When he or she specifies whether a Pattern is a part of the client or the server, he or she must have an idea about what the client-server architecture is. This is not always viable for modelers without technical background who are not rare in a game production team, especially when the target architecture becomes more complex, such as “Peer to Peer with Central Arbiter”. [99]

2) The abstraction level of the architectural elements added to the DSL is too low, which makes the DSL and the model bound to a concrete software architecture. E.g. when the Client-Server version of Orc’s Gold needs to be extended to a multiple server partitioning game to deal with larger game map, the RAIL language as well as the model must be revised. New properties need to be added and values of the properties need to be assigned for each Pattern. In a real project, it implies large amount of work.

To solve the above-mentioned problems, the abstraction level of the architectural elements has to be raised. The architectural information contained in the models should be more abstract than the generated code [60]. In next section I propose a method of modeling architectural characteristics at a proper abstraction level using GWG.

7.3 Use Game Worlds Graph (GWG) in MDGD

The GWG framework provides two abstraction layers: the Graph Style and the Flavor. The Graph Style concerns with the Game Worlds multiplicity and combination, while the Flavor concerns with the world connections. When the GWG is applied in the DSL
definition, it is also useful to split the concerns, because the Game World concept is closer to the game design domain, while the Connector concept is closer to the software architecture domain. If the two kinds of concepts are used in different part of the language, it is easier to assign the different part of the modeling tasks to modelers with different background, i.e. game designers and programmers.

In the GWG framework, three kinds of Game World were identified: Global World, Local World and Perceptible World. The Perceptible World is less relevant to game modeling, since the focus of the MDGD is mostly about the game state and behavior, instead of the presentation. The Global World and the Local World concepts are very useful because they distinguish the global state and the local state of the game. If we use these concepts to organize the elements of the game model, it is possible for the code generator to determine where the generated code for a game element should be deployed, and therefore the fundamental problem of supporting networked game architecture using DSL as shown in Figure 7.3 could be solved. Figure 7.4 illustrates how to combine GWG with game models in terms of the Game World concept.

Figure 7.4 Organize The Game Model with Game Worlds
The original model for standalone software architecture is organized as one package that the top-level element (the package) is the model itself. There are several elements contained in the package labeled as "Element A.1", "Element B.1", etc. When the model needs to be extended with integrating the GWG concepts, two sub-packages are introduced as the children of the root package, which represent the Global World and the Local World respectively. The model elements are now organized into the Global World and Local World packages according to the need to be shared between the players or only meaningful for one specific player. If it is a shared element it will be part of the Global World, and if it is not it will be part of the Local World. Therefore the model organized with Game Worlds contains the information about which elements need to be shared, which is important for generating expected architectural code. For example, a code generator for a Client-Server architecture can generate (part of) the server software from the model elements in the Global World package, and generate (part of) the client software from the model elements in the Local World package. In a more complex case, the model can contain multiple Global World packages, and each package contains the elements that have close logical relationship, e.g. they represent the game elements in the same region of the game world. A code generator for multi-server architecture can thus generate multiple server software in terms of the Global Worlds organization. Figure 7.5 illustrates how the multi-Global World game model is related to the target game.

Figure 7.5 Multi-Global World Game Model and The Target Game
The mapping between the Global Worlds and game regions in Figure 7.4 is not absolute; instead it could be decided by taking other concerns into account. For example, if the expected number of players is below a certain threshold, the game could be generated with a typical Client-Server architecture where all Global Worlds packages are generated into one server software package. If the number is above the threshold, a multi-server version could be generated according to a mapping strategy (e.g. as shown in Figure 7.5). The game model integrated Game Worlds concepts does not require the modeler having the knowledge about software architecture, and what they have to decide in modeling is that for each model element: If it should be shared by players and which part (e.g. region) of the virtual world it belongs to. The decision is relatively easy to make for game designers, since it does not involve technical concepts like "client", "server", "server farm", and "peer". The higher abstraction level of GWG actually makes the target architecture to be generated transparent to the modeler, which is one major advantage of the approach. Another advantage is that although the model contains the Game Worlds information, it is still not bound to a specific game architecture. The model in Figure 7.4 can be used to generate client-server architecture, multi-server architecture or even standalone software architecture depending on the configuration of the code generator.

There are at least three kinds of configurations for the code generator: 1) hard-coded configuration, 2) configuration file, and 3) model-level configuration. For the first method, we modify the code generation program to select the generated architecture. For the second method, a configuration file e.g. a XML file is used together with the game model by the code generator to determine which architecture to generate. Finally for the third method, the detailed information about the game architecture is specified within the game model when it is supported by the DSL. The first method is straightforward and easy to implement but difficult to modify. The second method is easy to modify, but difficult to understand the architecture since the architectural information is distributed between the model and the configuration file. The third method integrates the complete architectural information into the game model in cost of making the model architecture-specific. Choosing the method of configuration is dependent on the situation of the project, which requires comprehensive consideration that is outside the scope of this thesis. Here I will describe how to implement the third method of configuration in detail, because it will be used in the case study in the following chapter.

In order to configure the target architecture solely on the model level, the DSL must support two features:

1) The modeled game elements can be organized into the Game Worlds, including the Global Worlds and Local Worlds.

2) The connector types among the Game Worlds can be set in the model.

Feature 1) can be implemented by adding high level elements representing the Game Worlds in the model elements hierarchy, which is illustrated in Figure 7.5 for three types of concrete syntax.
To implement feature 2), the DSL can define several properties in the top model element corresponding to the connector types between different pairs of Game Worlds. For example, if we add two properties to Figure 7.6(a), representing the connector type between the Global Worlds and the connector type between the Global Worlds and the Local Worlds respectively, the resulted model will be something like Figure 7.6: two properties “GW_GW_Connector” and “GW_LW_Connector” were attached to the root element “Game”, whose values can be set in the model editor. If we use the model with the connector type information represented by the two properties to generate code, the code generator can therefore determine the architecture the code will be generated from according to the values of the properties. Figure 7.8 illustrates the code generation process.
Figure 7.7 Add Connector Type to A Game Model

Figure 7.8 Example of Code Generation Process Based on GWG

Figure 7.8 showed three possible combinations of the connectors, and the three generated architecture. Note that the way of using connector type with Game Worlds in
Figure 7.7 is one possibility, and in specific projects, the definition of the properties and their values can vary depending on the requirements to the DSL. In the case study presented in the next chapter, I only used one property with three possible values to generate code for three different architectures, which shows another possibility of using GWG in code generation. Integrating the connector type into the game model makes the model architectural-specific. This makes switching target architectures easier in cost of reducing the architectural portability of the model. Whether it is profitable is quite dependent on the requirements of the project.

7.4 Summary

The emergence of the Internet has brought many opportunities to the game industry, and most of today’s commercial games have online features. To support such features, the standalone software architecture is no longer sufficient for the modern games, and many advanced architectures have been developed as I have reviewed in Chapter 6. However, none of the MDGD approaches in the literature has addressed the diversity of architectures of computer games, and they have either targeted the standalone software architecture, or omitted the architectural aspects of the game, which constitutes a gap between the MDGD in academic research and the game industry.

To support various online game architectures in MDGD, the architectural concerns need to be addressed in modeling. To this end, the DSLs used must be capable of specifying the architecture information in game models. Therefore corresponding language constructs need to be defined, which requires proper abstraction of game architectures to reduce the gap between the software architecture domain and the computer game domain. The GWG framework presented in Chapter 5 provides useful concepts for defining the constructs, which lie on a higher abstraction level than the concepts from the general software architecture domain. The GWG framework provides two conceptual layers: the Graph Style layer and the Flavor layer. The former focuses on the Game World combination, and the latter focuses on the Connectors. When applying GWG in MDGD, the integration of GWG concepts with the DSL also include two similar parts: use the Game Worlds as packages to organize model elements, and use the Connector types to specify the concrete architecture to generate.

The use of GWG in MDGD provides an approach for including architectural information in game models, making it possible to support various game architectures with a single DSL. Moreover, GWG raises the abstraction level of game architecture, thus the approach has at least two advantages:

1) The Game World concepts are easier to understand than the concepts from the architecture domain for modelers without the technical background. Therefore integration the concepts to the DSL will not significantly affect the usability.

2) Porting the model with GWG information from one architecture to another is relatively easy.
In next Chapter I will extend RAIL to a DSL supporting several game architectures, and re-implement *Orc’s Gold* with three different architectures following the GWG-MDGD approach. I will further illustrate and evaluate the approach through a case study.
8  Case Study 2: Extend RAIL With Game Worlds Graph To Support Multiple Game Architectures

This chapter describes a case study based on the development and the evaluation of using the extended version of RAIL to develop the game *Orc’s Gold*. With integrating the Game Worlds Graph (GWG) concepts into the RAIL, the language is capable of including architectural information in the game models. With such information, proper infrastructural code can be generated for the target architecture. Three architectural patterns are currently supported by the extended RAIL: 1) traditional client-server, 2) integrated client-server, and 3) standalone software. An EMF-based tool-chain has been implemented to support the extended RAIL, which was then used in prototyping three versions of Orc’s Gold based on the above-mentioned three architectural patterns respectively.

The prototype was developed with both manual coding and using RAIL modeling, and the time spent and code lines written have been recorded and compared. Then results show that the extended RAIL approach can significantly save the development time and reduce the amount of manual coding in the game development. Further, the integration of GWG concepts and the RAIL makes it possible to support complex architectures in game modeling, while still keeps the model on high abstraction level, leaving technical details to the code generator and domain framework. Moreover, splitting the architectural concerns between different levels of details makes the model easy to port between architectures and the language flexible in handling diversity of architectures in games.

8.1 Purpose and Design of the Case Study

In Chapter 7 I presented a high-level approach for supporting diversity of game architectures in MDGD. The approach integrates concepts from the Game Worlds Graph (GWG) framework with game modeling languages to include the architectural information in the models. With such information, the code generator can generate the appropriate code for the target architectures executed on the domain framework supporting such architectures. The GWG framework is in an appropriate abstraction level for the software architecture and the game domain, enabling efficient architecture modeling. It is also easy to learn for people without technical background. In this chapter I will present a case study as the consecutive work of the GWG-based MDGD approach. The purpose of the case study is to validate the usefulness and the
effectiveness of integrating the GWG approach and game modeling. The results of this case study are from the development and the evaluation of the extended RAIL language and its tool-chain, as well as the revised *Orc’s Gold* game prototype.

The design of the case study is similar to Case Study 1, where the whole process included two phases: 1) develop a MDGD language and its tool-chain, and 2) develop a game prototype with the tool-chain and evaluating it. In the first phase, Reactive AI Language (RAIL) will be extended to support three game modes, which are client-server multiplayer mode, host-joiner multiplayer mode, and the original single-player mode. For this purpose, the expressive power of the language will be improved by integrating it with Game Worlds Graph (GWG) concepts. The integration will follow the approach presented in Chapter 7. The domain framework as well as the code generator will be modified correspondingly to support the new RAIL language. In the second phase, three versions of *Orc’s Gold* game will be created. They will be based on three different game architectures respectively, which are 1) traditional client-server architecture, 2) integrated client-server architecture, and 3) standalone software architecture. The game will be implemented with both using manual coding and using the MDGD approach. The data regarding time spent on development time and lines of manual code written will be collected and compared. Finally, the prototype will be evaluated based on the data.

### 8.2 Extend RAIL With GWG Concepts

As I have discussed in Chapter 7, GWG can be used in to design the game DSL, and the general idea is: 1) use Global Worlds and Local Worlds as packages to organize model elements, and 2) use Connector Type to specify architectural details. Following this approach, the RAIL meta-model can be modified as shown in Figure 8.1.

The classes in gray correspond to the Game World concepts including Global World and Local World. The AIPattern class is contained in these GameWorld classes instead of the Game class directly. The Game class owns the Game World classes. One property added to the Game class is the GW_LW_connector, which represents the Connector Type between the Global World and the Local World with the valid value set = {Local, Remote, Hybrid}. 
When modeling with the extended RAIL, the modeler will firstly create an instance of Game, and secondly add one GlobalWorld instance and one LocalWorld instance to the Game instance. Then the modeler can create AIPattern instances similarly to the models in the original RAIL approach. But the AIPatterns need to be created under (as part of) the GlobalWorld or the LocalWorld, depending on whether it influences the shared game state or only influence the local presentation of the game state. The target architecture also depends on the ConnectorType: if it is set to local, a standalone software architecture will be generated, if it is set to remote, typical client-server architecture will be generated, and if it is set to hybrid, integrated client-server architecture will be generated. Figure 8.2 illustrates an example of extended RAIL model and how the target game architectures are generated.

The game model in Figure 8.2 has a root element named ”TheExampleGame” whose type is Game, and it has two children named ”GW” and ”LW”, whose types are GlobalWorld and LocalWorld respectively. There are four AIPatterns named ”G1”, ”G2”, ”L1” and ”L2” in the model, which are organized into GW and LW according to their scopes of influence. As discussed before, TheExampleGame has a property named ”GW_LW_connector”, and it plays an important role in the code generation. The game model is used to generate the AI code for the target game by the RAIL Code Generator in Figure 8.2. The code generator is an extended version of the generator described in Chapter 4. It is not only able to generate the AI pattern code, but also able to generate the architecture-specific infrastructure code. In this case the latter is mainly the proxy
code for the synchronization of the AI pattern states between the client and the server. The code generator will use the value of GW_LW_connector to decide which architecture to generate, and if it is:

1) Local: All the pattern code will be put together, no proxy code is generated, and the domain framework will be configured as a standalone software architecture. Then the target game will be a single player game.

2) Remote: Two pieces of software will be generated, which are client-side software and server-side software. Code for G1 and G2 will be generated as part of the server-side software, while code for L1 and L2 will be generated as part of the client-side software. Proxy code will be generated on the client-side, which will receive the state update of G1 and G2 from the server-side, and controls the characters on the client-side on behalf of the G1 and G2. The domain framework will be configured as a Client-Server architecture. This means that the target game will be a client-server multiplayer game.

3) Hybrid: One piece of software will be generated with all the pattern code. Proxy code will also be generated, and whether it will control the character depends on the run-time mode of the software: If it runs as the host, the proxy code will not function, whilst the G1 and G2 code will control the character; and If it runs as the joiner, the proxy code will control the character whilst the G1 and G2 code will be disabled. The domain framework will be configured as integrated client-server architecture, and the target game will be a host-joiner multiplayer game.
Figure 8.2 An Example of The RAIL Model and Code Generation

The tool chain of the extended RAIL was also implemented on the Eclipse/EMF platform as it was done in Chapter 4 including the model editor and the code generator. The model editor is automatically generated from the modified RAIL meta-model, and the tree view is used as the concrete syntax. The code generator of the original RAIL needed a major revision carried out by adding decision logic and architecture-specific templates to the Acceleo sources. Torque 2D was still used as the game engine because it provides network features that is capable of implementing the target architectures. The manual code for integrating Torque 2D with a domain framework was also extended by adding architecture-specific infrastructure code, gameplay code, and GUI code. The code added to the domain framework was carefully organized, making it possible to configure the domain framework for different architectures in run-time.
8.3 The Prototype: Networked Orc’s Gold

The game concept of the Networked Orc’s gold is similar to the single player version: Players need to collect all the chests to win the game, and they will lose if the enemies killed all of them. There are at most two players in a game. They collect chests collaboratively, and share the same goal. Three versions of the game were developed based on the three architectures as we have discussed above where one was the single player version (functionally equivalent to Chapter 4), and the other two were the client-server version and the host-joiner version. For the client-server version, a dedicated server was created, which maintains the game state. A game client running on another machine can connect to the server and receive the game state update periodically through network. Figure 8.3 shows screenshots from the client-server version where the server’s waiting-for-connection UI is shown to the left, and the client’s joining-server UI shown to the right. The client is in charge of the game visualization and input handling, and the local game state on the client is only used for optimizing rendering and hiding latency. For the host-joiner version, hybrid software with both server and client functionalities was created. A player needs to choose between “Create a Server” or “Join a Server” when entering the game. If he or she chooses “Create a Server” the software will work as a server, and if he or she chooses “Join a Server” the software will work as a client. Once a player creates a server and another player joins, the game starts, and each player controls an avatar respectively. The game states are maintained on the server and periodically delivered to the client. Figure 8.4 shows screenshots of the user interface for choosing client/server and the in-game screen.

The AIPatterns created in Chapter 4 were reused in prototyping the Networked Orc’s Gold, by reorganizing them into the GlobalWorld package and the LocalWorld Package. The Guard, Dragon and Chest patterns were put into the GlobalWorld because they modify the game state shared by the players, while the Tree pattern was put into the LocalWorld because it is purely decorative and the distributed inconsistency of its state will not bring in any gameplay problems. Figure 8.5 shows the Networked Orc’s Gold model in the RAIL model editor. The details on the Trigger level and below are hidden to save space.
Figure 8.4 Screenshots of *Networked Orc’s Gold*: Host-Joiner Version

Figure 8.5 Networked Orc’s Gold RAIL Model
8.4 Evaluation of the Prototype

The Evaluation of the prototype follows the same methodology as the first prototype described in Chapter 4. The game Networked Orc’s Gold was implemented twice using the manual coding method and the MDGD method respectively. The time spent and the Lines Of Code (LOCs) for both methods were recorded and the results are presented in Table 8.1.

Table 8.1 Comparison of Development Investment with Two Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Used (Hours)</th>
<th>Manual LOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Domain Framework</td>
<td>AI Patterns</td>
</tr>
<tr>
<td>Manual Coding</td>
<td>35</td>
<td>13</td>
</tr>
<tr>
<td>MDGD with RAIL</td>
<td>0.8</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.1 shows that the Domain Framework consumed most development time, and also resulted in the most lines of code. This is because the Domain Framework implemented many technologies that are common to the genre, including low-level AI algorithms, input handling, network communication, and so on. I chose to manually implement these technologies as part of the domain framework because they are on a low abstraction level thus appropriate for the General Purpose Language (GPL) programming. Also note that because the Domain Framework is on a low abstraction level, it can be reused when new high-level AI is added, or new games are developed, which leverages the one-time investment on its development.

Similarly to the first prototyping, the development time consumed on AI Patterns were dramatically reduced when the MDGD was applied. For the manual coding method, it took 13 hours to develop the AI Patterns, which is 4.5 hours more than the non-networked, first prototype. The extra time was invested on implementing the synchronization mechanisms of the patterns between the client and the server. When it comes to the MDGD method, it took only 0.8 hours (48 min) to model the AI Patterns, which is only 0.1 hours (6 min) more than modeling the non-networked version. Thanks to the GWG framework, the extended RAIL elegantly supports specifying the networked aspects of games by adding only three language constructs (Global World, Local World and Connector) to the original RAIL. Extending a non-networked RAIL model to networked-architecture only requires three steps as illustrated in Figure 8.6: 1) create a Global World and a Local World element; 2) drag and drop AI Patterns into a proper Game World; and 3) set the GW_LW_connector property in the Game element. The LOC data also explains the difference of the development time: 1133 manual code lines were saved when MDGD was applied.
Figure 8.6 Three Steps Extending RAIL Model to Networked Architecture

The productivity gain is not free, and an initial investment on the DSL and the tools was needed as presented in Table 8.2. However it is relatively low: even if we add the time and LOCs to the game development cost, we still achieve productivity profit of 2.7 hours and 150 LOCs. Furthermore, if we consider the potential reuse of RAIL tools, the productivity improvement from applying MDGD is even more significant.
Table 8.2 Cost of Developing Extended RAIL and Its Tool-Chain

<table>
<thead>
<tr>
<th>Extended RAIL and Its Tool-Chain</th>
<th>Time (Hours)</th>
<th>LOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9.5</td>
<td>983</td>
</tr>
</tbody>
</table>

The workflow of RAIL modeling in the prototyping is natural and efficient: the process of modeling AI Patterns is similar to the non-networked RAIL, and the additional workload for supporting three architectures is minor, including:

1) Deciding which Game World a particular AI Pattern belongs to, and this depends on whether the state of the AI Pattern is shared by all players participating in the game.

2) Set the GW_LW_Connector to a specific value to generate code for a corresponding architecture.

The RAIL used Game World and Connector concepts to split two levels of architectural concerns, so that the modeler of AI Patterns does not have to know the details about the target architecture. The modeler only needs to decide on which Game Worlds the AI Patterns belong to. Therefore the programmer/engineer can set the connector type after the modeling is finished, which finally determines the architecture of the generated code. The advantage of this approach is that non-technical people such as game designers do not need the knowledge of game architecture thus lowered the threshold of game AI development, which is usually done by programmers in traditional game development.

Splitting the architectural concerns resulted in another advantage on workflow that the AI Patterns in the RAIL model are architecture-independent. Porting the patterns from one architecture to another only needs to change one property (the GW_LW_Connector). The development of Multiplay Orc’s Gold has benefit from it that the patterns of the game was initially developed in standalone software architecture supporting only single-player mode, and when the game is almost finished and fine tuned, I port it to a networked architecture, which took only several minutes. Debugging and tuning in standalone software architecture is much more easier than in networked architectures, so the workflow is optimized thanks to the easy switching of architectures supported by the RAIL.

8.5 Summary

In this chapter I modified the RAIL and its tool-chain to support three game architectures, which are standalone software, traditional client-server and integrated client-server. The modification to the RAIL follows the approach presented in Chapter 7, and the modified tools were used to develop the prototype Networked Orc’s Gold.

Thanks to the GWG framework, the RAIL now has the expressive power to address the architectural concerns, with which modelers can insert the architectural information into the high-level AI models that enables the code generation for different game architectures. The prototyping of Networked Orc’s Gold shows that the RAIL tools 1)
significantly improve the productivity in the target domain, and 2) optimize the workflow of game development. The RAIL splits the low-level architectural information (connector-type level) from the high-level architectural specification (Game World level), so that the developers of high-level AI do not have to know the details of the target architecture thus the threshold of development is lowered, allowing non-technical game developers develop the high-level AI even for networked games. Moreover, changing the target architecture only needs to change one property in the model, and this improves the architectural portability on one hand, and allows taking the advantages of different architectures in different development phases on the other.

As I have discussed in Chapter 2, none of the existing MDGD approaches in literature has addressed architectural concerns. To my best knowledge no networked game or prototype has been developed before in the MDGD community, so the extended RAIL and Networked Orc’s Gold are valuable contributions in this field. Further, the case study illustrated how the approach proposed in Chapter 7 is technically implemented, and how it works in practice. Thus validating the usefulness of GWG framework is also a major contribution of this chapter.

The extended RAIL supports three game architectures, however Chapter 6 has showed that the GWG framework is capable of classifying and analyzing many more game architectures. Thus the potentials of the GWG framework in helping MDGD still require further exploration. Moreover, MDD itself also has many possibilities when applied to game development. Integration of the GWG and MDGD can open a broad prospect of future research.
9 Evaluation

This chapter evaluates the research documented in the thesis. The evaluation will be carried out from three perspectives: 1) the answers to the research questions, 2) the methodology, and 3) the contributions and publications. For the first perspective, I will review the research questions described in Chapter 1, and answer them based on the results from my Ph.D. research. For the second perspective, I will make a structured discussion about the research method and process following the guidelines for Design Science in information systems research suggested by [17]. Finally, for the third perspective I will summarize the contributions made through my research, including the artifacts developed and the papers published as well as planned.

The chapter also includes a discussion about the limitations of the research, and a short summary of the thesis.

9.1 Answers to Research Questions

In Section 1.3, two major research questions were identified, which are derived from the overall research goal to reduce the two gaps remaining between MDGD research and industrial game development. The two major questions were further decomposed into six sub-questions. In this section I will answer them one by one according to the results from the Ph.D. research.

9.1.1 Answer to RQ1

RQ1 was defined as “How Shall MDGD be integrated with game engine-based development?”
To answer the question, a MDGD approach named Engine-Cooperative Game Modeling (ECGM) was designed and described in Chapter 3. The approach bridges the model-driven game development and engine-based game development, minimizing the cost of transition from the traditional method to the MDD method. Further, to evaluate the feasibility and usefulness of ECGM in practice, the domain-specific language RAIL and its tool-chain were created, and they were utilized in developing the prototype game Orc’s Gold. The prototyping is documented in Chapter 4.

RQ1 was decomposed into two sub-questions, and they were answered in Chapter 3 and Chapter 4 respectively. I will summarize these answers in the remaining of this section.
**RQ1.1: What is a general model-engine integrative approach that is not restricted to a particular engine or model-driven tool?**

ECGM as a high-level game development approach can answer this question. The basic idea of ECGM can be summarized as: 1) Use model-driven development for high-level game elements that are closer to the problem domain; 2) Use game engine script or native language code to development low-level game elements; 3) Use game engine tools to develop level data; and 4) Use Code-generation/Model-transformation technologies to integrate the artifacts from the traditional method and the artifacts from the MDD method, by transforming the models into both code and level-data.

As I have discussed in Chapter 2, existing MDGD approaches has either overlooked the game engine, or used it only as a run-time library. However, ECGM respects the dominance of engine-based development in practice, and embraces the game engine by reusing its libraries and tools, while still takes the advantage of model-driven development by letting the models do what they are suitable for. Code-generation/Model-transformation is the heart and soul of this approach, which makes the artifacts from two development activities interoperable. The reuse of run-time engine and engine tools reduce the scope of MDD, thus lowered the cost of creating DSL tools and minimized the modifications to the traditional development process. Moreover, the investment on learning game engine tools is preserved, which can potentially reduce the resistance of applying MDGD from the organizational perspective.

**RQ1.2: How should we implement an integrative approach with a specific engine and MDD tool?**

To answer the question, the Reactive AI Language (RAIL) and its tool-chain were developed, and they were integrated with the commercial game engine Torque 2D. The integration followed the ECGM approach where both engine script code and level data can be generated from the RAIL models.

The RAIL is a state-machine-like DSL with tree-view as the concrete syntax, which was designed for modeling high-level AI in Action/Adventure games. Torque 2D is a commercial 2D game engine with the fully-featured world editor Torque Game Builder (TGB). Torque 2D supports a script language named Torque-Script, which is a c-like language with some (not full) object-oriented features. The RAIL tool-chain includes model editor, code generator, and semantics validator. The RAIL tool-chain is based on Eclipse/EMF framework, and the model editor is automatically generated by the framework. The code generator is created with Acceleo, which is a code generation framework on Eclipse platform. The code generator can both generate Torque-Script code feeding the run-time engine, and the level data processable by the TGB.

To evaluate the usefulness of the RAIL tools, the prototype Orc’s Gold was developed, which is a single-player adventure game. The evaluation of the prototype revealed that the RAIL tools can significantly improve the productivity in the target development environment, and enables an efficient workflow. The implementation of the RAIL tools and the prototype validated the feasibility and usefulness of ECGM, and answered the research question.
9.1.2 Answer to RQ2

RQ2 was defined as “How Should The Architectural Diversity Of Games Be Addressed in MDGD?” The question was addressed because software architecture plays an important role in developing computer games, which will also influence MDGD in many aspects. The diversity of game architectures has been an important motivation for the research from game community, and many architectures have been proposed, which need to be supported by MDGD.

To answer the question, several inter-related research activities were carried out, addressing different aspects of the question. The answer to RQ2 is centered on the Game Worlds Graph (GWG) framework, which is a conceptual framework for classifying, analyzing, and describing existing game architectures, and also for exploring future game architectures. The GWG framework was therefore used as the taxonomy framework for reviewing game architectures in literature to analyze the current situation of game architecture research. The results of the review provided the background information for the next step of the research, where a MDGD approach was developed based on GWG, addressing game architectural supports in Domain Specific Languages (DSLs). To validate the feasibility and usefulness of the approach, the RAIL was extended to support three game architectures following the approach, and then it used to develop three versions of the game Networked Orc’s Gold with different architectures.

The results from all the above-mentioned research constituted the answer to RQ2. In the rest of the section, I will elaborate the results by answering the four decomposed sub-questions of RQ2.

**RQ2.1: What is the current situation of the architectural diversity of games?**

The answer to the question is documented in Chapter 6 where I reviewed 40 game architectures in the literature, and analyzed their characteristics using the GWG framework. The architectures focused on various quality attributes, such as load balancing, lowering latency, and anti-cheating. The complexity also varies, from standalone software to multiple-server. Most of the architectures were created for networked games, especially for MMOGs, confirming that supporting networked architectures is important for MDGD.

**RQ2.2: What is an appropriate conceptual framework of game architecture that is useful in MDGD?**

The Game Worlds Graph framework presented in Chapter 5 answered this question. The central concepts of the GWG framework are Game Worlds and Connectors. On top of them, Graph Style and Flavor were defined. The GWG framework provides new perspectives to game architectures with a higher abstraction level than the perspectives from general software architecture domain. GWG can be used to classify and analyze existing game architectures, which was proven by the literature review in Chapter 6.

Since the GWG framework revealed essential characteristics of game architectures, it could also be used in MDGD. How has GWG worked with MDGD will be described in the answer to the next research question.
RQ2.3: How can the architectural concerns be addressed with a MDGD approach based on the conceptual framework?

The answer to the question was documented in Chapter 7, where the GWG framework was used in MDGD to achieve two goals: 1) integrating the architectural information in the modeled game elements, and 2) determine the target architecture to generate the code for. A MDGD approach was therefore proposed, which: 1) used the Global World and Local World concepts in game DSLs for organizing the modeled elements according to their scope of influence, and 2) use the Connector Type concept to determine the target architecture.

The approach firstly showed a way of supporting networked architectures in MDGD, and secondly allowed to port the game model from one architecture to another without major modifications to the model, which is by the way very difficult with traditional coding method.

RQ2.4: How can we implement a specific modeling language and tool chain addressing game architectural concerns?

This question was answered in Chapter 8, where the RAIL and its tool chain were extended for addressing architectural concerns. The Global World and Local World concepts were used in RAIL to package AIPatterns. If an AIPattern influences the shared game state, it will be included in a Global World, else it will be included in a Local World. A property corresponding to the Connector Type between the Global World and the Local World was defined, and its value determines the target architecture that can be standalone software, client-server, or integrated client-server.

The extended RAIL tools were created using Eclipse/EMF framework and Acceleo. A prototype with three possible architectures were developed and evaluated. The results showed that the extended RAIL tools based on GWG can significantly improve the productivity in the target domain, and they can also enable an efficient workflow where game developers with different background can focus on different parts of the model, and switching between game architectures is simple and easy.

9.2 Methodology

Design Science was used in the research as the main research method. Design science research has been practiced in Computer Science, Software Engineering and Information Systems for decades [152]. The fundamental principle of design-science research is that knowledge and understanding of a design problem and its solution are acquired in the building and application of an artifact [17]. My Ph.D. research is coherent with the principle, where artifacts such as conceptual framework, DSM Tools and prototypes were created, through which research questions were answered. von Alan further proposed seven guidelines to depict effective design science research [17]. I will discuss my research according to the guidelines, in order to evaluate the quality of the research method.
**Guideline 1: Design as an Artifact**

The guideline pointed out that the result of Design Science research should be an IT artifact, and it must be described effectively, enabling its implementation and application in an appropriate domain [17]. The importance of IT artifact were also emphasized in [150, 152].

Artifact was defined as *constructs, models, methods* and *instantiations* in [17]. The results of my research comply with the definition. The major artifacts described in the thesis include:

1) The Game Worlds Graph (GWG) framework: GWG is both a *model* and a set of *constructs*. As a conceptual model, GWG defined a set of concepts and their relationships, revealing the essential characteristics of game architectures. It also provides terms such as Global World, Local World, Remote Connector, 101, 1nn, which constitute a vocabulary for describing the game architecture domain.

2) The Engine Cooperative Game Modeling (ECGM) approach: ECGM is an engineering *method* for effectively and pragmatically integrating MDGD into engine-based game development.

3) GWG-based Game Modeling approach: The approach is also an engineering *method*, which supports specifying architectural information in game models, thus enables generating code for various game architectures. With the approach the architectural diversity of computer games could be addressed in MDGD.

4) RAIL and extended RAIL: They are *instantiations* of 2) and 3). As the MDGD tools implementing ECGM and GWG-based modeling, they have been successfully used and evaluated in two case studies, with which the feasibility and usefulness of the methods were demonstrated. The tools provided “proof by construction” [7].

**Guideline 2: Problem Relevance**

The objective of research in information systems is to acquire knowledge and understanding that enable the development and implementation of technology-based solutions to heretofore unsolved and important business problems [17]. The Ph.D. research has followed the guideline by addressing two unsolved and important problems in the MDGD field that are 1) how MDGD can be integrated with Game-engine based development, and 2) how MDGD can support various game architectures, especially networked-architectures.

Moreover, the relevance of any design-science research effort is with respect to a constituent community, to be relevant with which, research must address the problems faced and the opportunities afforded by them [17]. The constituent community of MDGD is the practitioners in the game development field. As I have discussed in Chapter 2 and Chapter 6, the engine-integration and architectural diversity are both important problems for the practitioners, so the research is also relevant to its constituent community.
Guideline 3: Design Evaluation

The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well executed evaluation methods [17]. The different artifacts of the research have been evaluated using different methods.

For the GWG framework, since it is a conceptual framework, which can derive many applications, it was evaluated with multiple methods: The overall quality was evaluated analytically in Chapter 5 together with the quality of exploring new architectures; Its quality of classifying and analyzing game architectures was evaluated by a systematical review in Chapter 6; and Its usefulness in MDGD was evaluated by instantiating DSL tools and a prototype in Chapter 7 and 8.

For the ECGM (Chapter 4) and the GWG-based MDGD (Chapter 7), they were both evaluated through instantiating concrete DSL tools, and the prototyping. The DSL tools demonstrated how the methods could be technologically implemented, and the prototyping experimentally evaluated the usefulness and effectiveness of the methods.

For the RAIL and networked-RAIL tools, they were both evaluated through prototyping games. The data regarding development effort and performance proved the usefulness of the tools, and the user experience during the prototyping confirms the efficiency of workflow.

Guideline 4: Research Contributions

Interesting contributions justify the value of research. In [17], three kinds of contributions of design-science research were identified, which are the design artifact, foundations, and methodologies. The Ph.D. research contributed to all three aspects: the design artifacts have been discussed before; GWG is a kind of foundations for game architecture related research; and two MDGD approaches contributed to game development methodology. The details of the contributions will be summarized in Section 9.3.

Guideline 5: Research Rigor

Rigor addresses the way in which research is conducted, and design-science research requires the application of rigorous methods in both the construction and evaluation of the designed artifact [17].

The research was carried out with rigor always in mind:

1) For the construction of the artifacts:
   a. Graph Theory was used to ground the GWG framework;
   b. All mathematically possible Graph Styles were examined to identify viable Graph Styles;
   c. Formal meta-models were used in constructing RAIL and net-worked RAIL;

2) For the evaluation of the artifacts:
a. Systematic Review was used to evaluate the usefulness of GWG framework as a taxonomy framework for game architectures. The method is more rigorous than normal literature review.

b. Experiment method was used in evaluating the prototypes regarding the productivity and performance.

However, there is still some rigorous flaws in the research, such as the workflow evaluation of the MDGD approaches was based on user experience lacking the mathematic formalism, the evaluation of GWG for exploring new architectures was only done by analyzing the method and illustrating an example, the subject groups of the experiment were restricted to one developer. Some of the flaws can be explained by “an overemphasis on rigor can lessen relevance [17]”, others however have to be pointed out in Section 9.4 as the limitations of the research.

**Guideline 6: Design As A Search Process**

Design is essentially a search process to discover an effective solution to a problem, and the search requires utilizing available means to reach desired ends while satisfying laws in the problem environment [17]. The Ph.D. research did follow the principle to create the artifacts. For some of artifacts, it is possible to search the entire solution space, for example, to identify viable Graph Styles, I searched all possible combinations of multiplicities of Game Worlds.

However, as it was pointed out in [40], given the wicked nature of many information system design problems, it may not be possible to determine, let alone explicitly describe, the relevant means, ends, or laws. Even when it is possible to do so, the sheer size and complexity of the solution space will often render the problem computationally infeasible [17]. In the Ph.D. research, such situation was common. For example, it is not possible to search all possible DSL design blueprints that work for Action/Adventure game modeling to find the best solution. In such situations, the search was done only for satisfactory solutions the designer knows with the input from other experts if possible. This is also suggested by [41] and [17].

**Guideline 7: Communication of Research**

Design-science research must be presented both to technology-oriented as well as management-oriented audiences [17], and they have different concerns.

The thesis presented the research to technical people by providing the sufficient details about:

1) Processes and examples about how to use GWG to analyze, explore, and classify game architectures;

2) Technical architecture and examples about how to implement ECGM and GWG-based game modeling; and

3) Meta-models and technical specifications about how to create and use RAIL and extended RAIL tools.
This enables practitioners to apply the artifacts in game projects and researchers to build a cumulative knowledge base for further evaluation and extension.

The thesis also presented the research to management people by providing the details about:

1) What modifications to the existing technical architecture are needed when applying ECGM;
2) The investment on learning game engine tools could be protected; and
3) The cost and profit from applying the MDGD approaches and the meta-tools useful in MDGD.

This information is important for the management-oriented audience to make the decision about applying the MDGD approaches in their projects.

9.3 Contributions and Publications
The major contributions of the research are presented in Table 9.1 together with the publications and planned publications. Because the thesis is a “monography” instead of a “paper collection”, most of time was invested on solely writing the thesis and thus the number of publications so far has been low. However, the unpublished contributions in Table 9.1 at least justify three articles that were already planned. After the thesis is submitted, the articles will be extracted from the thesis and submitted.

<table>
<thead>
<tr>
<th>Contribution</th>
<th>Description</th>
<th>Publication/Planned Publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. GWG Framework</td>
<td>GWG Framework is a conceptual framework for game architectures. Its application includes classifying and analyzing existing game architectures, exploring new game architectures, providing conceptual support for MDGD, and so on.</td>
<td>[153]</td>
</tr>
<tr>
<td>2. ECGM Approach</td>
<td>ECGM is a MDGD approach featuring the integration with the game engine including its tools. Code generation/model transformation played an important role in bridging the MDD and engine-based development methodologies.</td>
<td>[A1]</td>
</tr>
<tr>
<td>3. GWG-based Game Modeling Approach</td>
<td>The approach is a solution for addressing architectural diversity in game modeling. Thanks to the descriptive power of the GWG framework, the approach can enhance</td>
<td>[A2]</td>
</tr>
<tr>
<td>No.</td>
<td>Section Title</td>
<td>Content</td>
</tr>
<tr>
<td>-----</td>
<td>---------------</td>
<td>---------</td>
</tr>
<tr>
<td>4.</td>
<td>RAIL and its tool-chain</td>
<td>DSLs with supporting various game architectures, and the target architecture can even be switched easily during development.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RAIL and its tool-chain is firstly an instantiation of the ECGM approach, and secondly a DSL tool suit for developing Action/Adventure games. The contributions of RAIL include the methodological aspect as well as the technological aspect.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[A1]</td>
</tr>
<tr>
<td>5.</td>
<td>Extended-RAIL and its tool-chain</td>
<td>Extended-RAIL and its tool-chain is firstly an instantiation of the GWG-based modeling approach, and secondly a DSL tool suit for developing networked Action/Adventure games.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[A2]</td>
</tr>
<tr>
<td>6.</td>
<td>A Literature Review of MDGD</td>
<td>The review is a state-of-art investigation of the MDGD domain. Representative work was identified and discussed, and gaps were pointed out.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[A3]</td>
</tr>
<tr>
<td>7.</td>
<td>A Systematic Review of Game Architecture</td>
<td>The systematic review firstly validated the usefulness of the GWG framework for classifying and analyzing game architectures, and secondly provided a comprehensive view of game architecture research.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[154]</td>
</tr>
</tbody>
</table>

The most important publication in the Ph.D. research so far is [153], which presented the Game Worlds Graph framework and was published by a top journal in the game field (ACM Computers in Entertainment). Apart from the already published two journal articles [153] and [154], three articles were planned in Table 9.1, where [A3] will be a literature review paper on the MDGD domain, [A1] will be a paper describing the ECGM approach and use RAIL as the instantiation, and [A2] will be a paper describing the GWG-based modeling and use extended RAIL as the instantiation.

### 9.4 Limitations of The Research

Apart from the achievements and contributions, the Ph.D. research has some limitations mainly in the methodological and technological aspects. I will point out the major limitations in this section.

Although the ECGM approach has shown its usefulness in the RAIL and its tools, there is only one instantiation of the approach so far. More instantiations, i.e. more DSLs and game engines should improve the reliability of the evaluation of the approach, which however did not fit into the timeframe of my Ph.D. research. The same limitation exists for the GWG-based game modeling approach.
The RAIL and extended RAIL tools achieved impressive results on the productivity gains in prototyping Orc’s Gold and Networked Orc’s Gold. However, the games developed were still proof-of-concept prototypes instead of complete games. If several real games are developed using the tools, it will greatly improve the fidelity of practitioners on MDGD. To develop even one complete game is not a simple project, whose budget exceeds the resources of a Ph.D. project.

The subject group of the experimental development of the prototypes is limited to one developer, which threatens the rigor of the evaluation. If more resources were allocated to the Ph.D. project, it would have been evaluated in a more reliable setting.

The above-mentioned limitations mainly result from the budget of the Ph.D. project. The deep reason behind it is that the Ph.D. research spanned over Model-Driven Development, Game Development and Software Architecture Domains, and the cross-field nature of the research implies a broad problem and solution space, which requires much more resource than the Ph.D. project to exhaustively explore. On the other hand, the industrial game development itself is complicated and has the cross-disciplinary nature, and to achieve the research goal, the situation is not avoidable. For the sake of the limitations, the gap between the MDGD in academic research and its industrialization still need effort from many researchers and practitioners to close, and the thesis opened a door to future research.

9.5 Summary
This chapter evaluated the Ph.D. research from three perspectives. The results showed that the observations through the research have answered the research questions, in terms of which the research goal has been achieved. The main research methodology used in the research was Design Science, which was evaluated according to the seven guidelines proposed in [17]. All the seven guidelines were addressed to some extent in the research, showing that the method of the research was proper and effective. The own contributions of the research include one conceptual framework (GWG), two MDGD approaches, and two DSLs and tool-chains. Apart from the own contributions, two literature reviews were done which resulted in a state-of-art report of MDGD and a report on game architecture domains. Two journal articles have been published, and three more articles are planned.

The limitations of the research was also pointed out, including more instantiations of the approaches are desired for improving the reliability of the methodologies, more games are desired for validating the usefulness of the DSL tools, and more rigorous experiment design can improve the quality of evaluation. The limitations were resulted from the cross-field nature of the research, whose problem space requires much more resource than the Ph.D. project to exhaustively explore. However the contributions to various fields still justify the sufficiency of the research within the time frame, and the limitations motivate future research, which will be presented in Chapter 10.
10 Conclusion and Future Work

Firstly, we take a look back at the research goal presented in Chapter 1:

To push the Model-Driven Game Development methodology further towards the practical use in commercial game development, by proposing methodological and technical solutions to reduce two gaps: 1) the gap between the MDGD and engine-based development, and 2) the gap between the existing MDGD approaches and the diversity of game architectures.

After the research has been presented, I can conclude that the goal has been achieved, and the conclusion is supported by the results and contributions of the research:

1) A prototype game Orc’s Gold has been developed using an approach integrating the MDGD and engine-based development. The approach has been described in detail, and been instantiated by integrating the RAIL DSM tools and the full-scale commercial game engine Torque 2D. To my best knowledge there is no existing MDGD approach that has cooperated with a commercial game engine or its tool suit before. I conclude therefore that the first gap has been reduced.

2) A prototype game Networked-Orc’s Gold has been developed using MDGD, which supports three game architectures including two networked architectures. The MDGD approach for supporting various game architectures were described, and instantiated using the Game Worlds Graph (GWG) conceptual framework presented also in the thesis. There is no existing MDGD approaches in literature that has addressed the diversity of software architectures of computer games before, so my research has also reduced the second gap.

However, in the road towards MDGD industrialization, challenges still remain. The work done in the Ph.D. research is still in academic domain, where the problems the game industry faces were abstracted and simplified. For example, the complexities of a full-scale commercial game and a prototype are different, which will influence the effectiveness of MDGD. Moreover, a cross-disciplinary team with 100 members also works very differently than a one-man team. Actually, this is a common problem with Design Science research, as it was pointed out in [17]: “Design Science research often simplifies or decomposes a problem, which may not be realistic enough to have a significant impact on practice but may represent a starting point.” I hope the thesis could serve such a starting point. A starting point allows the practitioners in game industry stand on and take a serious consideration about whether and how to introduce MDGD to their process.
Moreover, as a former game developer and a present researcher, I have some comments to MDGD in addition to the conclusion of the research. They are personal experience and reflections, instead of rigorous propositions. However I believe they are still useful for both researchers and practitioners:

1) *Give only to the MDGD the things that fit MDGD, and leave the rest to coding.*
There are some elements of game natively belonging to manual coding, because the engine-script or GPL also have their appropriate domains. If it is difficult to decide the scope of MDGD, rather narrowing it than broadening it.

2) *Model-level debugger is crucial for an efficient workflow.* If it is not ready, close communication between tool programmers and modelers must be enabled. The tools especially the code generator will never be bug-free in practice. Debugging in the generated code, and fixing in the code generator is even more difficult than with manual coding. Level designers may be capable of modeling, but they can hardly be in charge of non-model-level debugging.

3) *Try not to start MDGD from scratch.* Take a reference DSL tool as the start point. If it is not possible, use purely manual coding in pre-production, while creating the DSL tool at the same time. The initial investment of a DSL is high, which can delay the whole development process if it is on a critical path. Working sample code can significantly accelerate the development of code generator, which could be produced in pre-production.

In the rest of the chapter I will discuss future work.

**Future Work**

I have discussed the limitations of the research in Chapter 9, and some future work has been planned or will be planned to address the limitations:

1) The ECGM approach and the GWG-based modeling approach need to be instantiated by more DSL tools with more engines. The DSLs should focus on various genres respectively, especially on the mainstream genres.

2) The design of the experimental development could also be refined with involving more developers, and more projects. This will improve the rigor of the research. If cross-disciplinary teams could be used in the experiment, the impact of the MDGD approaches on workflow could be better understood.

3) The GWG framework still has high potentials that were not explored in the research. The framework can analyze and classify all 40 game architectures in the literature review, and it is however only used for generating code for three game architectures. The opportunities and limitations of the GWG in MDGD are worth future explorations.

4) RAIL tools can be extended with adding more language elements, in order to improve its expressiveness. Currently, it generated 1/6 of the total code lines, but there are some elements that ought to be modeled were not modeled due to the limitation of RAIL, such as player controller, global rules. Incorporating these elements into the RAIL model is also left for future work.
The thesis has answered several fundamental questions on the MDGD industrialization from where we can look forward to the future research directions of the MDGD field. The remaining part of this section will point out some major research questions that can inspire future work.

**FQ1: What can MDGD contribute to a commercial game project?**

Many publications including this thesis have reported significant productivity improvements from applying MDGD, however they are all based on prototypes. An important milestone on the road to MDGD industrialization will be the evaluation of using MDGD methodology and technology on full-scaled games. Since industrial games are much more complex than research prototypes, it may help reveal more advantages or limitations of MDGD. For example, in the *Orc’s Gold* prototype, I have only implemented four AIPatterns. Although the RAIL based modeling has reduced the development effort by modeling the four AIPatterns, the productivity gains can hopefully scale up with the increasing number of AIPatterns. On the other hand, more AIPatterns may require the DSL to be more complex, which can increase the cost of developing the DSL tools. The influence of the game complexity is therefore interesting for MDGD evaluation.

It is also obvious that the data from MDGD for industrial projects can easier convince the management of game industry. However, without the acceptance of the management, it is hard to try MDGD in commercial game development, and without the evidence from full-scale game development it is hard to persuade the management to embrace MDGD. Thus the problem becomes “which came first, the chicken or the egg?” My suggestion is that the researchers in the game research community have to break the circle by incrementally extending prototypes to full-scale games, so that the practitioners will have the confidence to follow-up.

**FQ2: How will MDGD change the organization of the game development team?**

Real game development teams are cross-disciplinary, and can be large. The organizational aspects such as roles and responsibilities, communications and co-operations are crucial for the success of game projects. The influence of MDGD to the organizational aspects of game development is therefore an interesting direction to explore.

MDGD releases programmers from some manual programming work, but it introduces two new questions regarding the team organization to answer:

1) Who is the modeller? The answer can be the *programmer*, the *game designer* or a *new role* to be introduced. The technical difficulty of modelling can be different for different DSLs and projects, so the answers will also be different. The ideal scheme may be that the game designer solely model the game elements with the help of mature DSL tools, however this may not always applicable in a real project.
2) Who is the MDGD tool developer? The answer can be the *gameplay programmer*, the *engine programmer*, the *tool programmer* or a *new role*. Developing a DSL and corresponding tools is not an easy task, which requires the knowledge with the DSL workbenches, and the skills on system architecture and problem abstraction. Current game development teams in the industry do not have the expertise so recruitment and/or training is needed.

The game industry used many years to commit to an efficient team structure, and they now have to adapt themselves to the MDGD methodology, which will not be easy, and the data from preliminary research will therefore be very interesting for the management of the industry.

**FQ3: What modifications to the game development process are needed to adapt MDGD?**

MDGD introduces new artifacts to the game development: the DSL tools, the domain framework, the models, and the generated code. The creation and maintenance of the artifacts needs to be merged into the game development process. New activities have to be introduced and coordinated. The dependencies between the activities and between them and the traditional activities have to be considered and evaluated to create an efficient MDGD process. For example: 1) modeling and corresponding code generation are dependent on the DSL tools, so there must be a working DSL before implementing gameplay; 2) the domain framework must be created before executing the generated code; and 3) the DSL development requires reference code to be efficient.

The thesis mainly discussed the technological aspects of MDGD industrialization, and the process aspects were not included, so the question remains for future research. From my point of view, most of the DSL tools and domain framework development should be done within the pre-production phase, and during the production phase the MDGD will focus on the modeling activity. However considering the iterative nature of game development, this may be difficult. As we can see most of the development tools have to evolve throughout the project to support changing game design, we cannot expect the requirements to MDGD tools to be unchanged. The adaptation of MDGD to the game development process is a direction worth further exploration.
BIBLIOGRAPHY

<table>
<thead>
<tr>
<th>Reference</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>35.</td>
<td>Mernik, M., J. Heering, and A.M. Sloane, *When and how to develop domain-</td>
</tr>
</tbody>
</table>


42. Altunbay, D., E. Cetinkaya, and M.G. Metin, Model Driven Development of Board Games, in the First Turkish Symposium on Model-Driven Software Development (TMODELS). 2009.


46. Pleuß, A., Modeling the User Interface of Multimedia Applications Model Driven Engineering Languages and Systems, 2005


62. Tolvanen, J.-P. and S. Kelly, *Defining Domain-Specific Modeling Languages*


75. Havey, M., Essential business process modeling. 2005: " O'Reilly Media, Inc.".

76. Kent, S. Model driven engineering. in Integrated formal methods. 2002: Springer.


79. Indraprastha, A. and M. Shinozaki, The investigation on using Unity3D


