A MODEL-DRIVEN FRAMEWORK TO SUPPORT GAMES DEVELOPMENT: AN APPLICATION TO SERIOUS GAMES

STEPHEN TANG OON THEAN

A thesis submitted in partial fulfilment of the requirements of Liverpool John Moores University for the degree of Doctor of Philosophy.

July 2013

ABSTRACT

Model Driven Engineering (MDE) is a software development approach which focuses on the creation of models to represent a domain with the aim of automatically generating software artefact using a set of software tools. This approach enables practitioners to produce a variation of software in by reusing the concepts in the domain model without worrying about the technical intricacies of software development. Therefore, this approach can help to increases productivity and it makes software design easier for the practitioners. The application of this approach into games development domain presents an interesting proposition and could help to simplify production of computer games.

Computer games are interactive entertainment software designed and developed to engage users to participate in goal-directed play. Many find computer gaming to be persuasive and engaging, and they believe that through the application of game design and game technology in non-entertainment domains can create a positive impact. Computer games designed primarily for non-entertainment purpose are generally known as serious games. The development of games software, in no relation to the intended purpose of it, is technically complex and it requires specialist skills and knowledge. This is the major barrier that hinders domain experts who intend to apply computer gaming into their respective domains. Much research is already underway to address this challenge, whereby many of which have chosen to use readily available commercial-off-the-shelf games while others have attempted to develop serious games in-house or collaboratively with industry expertise. However, these approaches present issues including appropriateness of the serious game content and its activities, reliability of serious games developed and the financial cost involved. The MDE approach promises new hopes to the domain experts, especially to those with little or no technical knowledge who intend produce their own computer games. Using this approach, the technical aspects of games development can be hidden from the domain experts through the automated generation of software artefact. This simplifies the production of computer games and could provide the necessary support to help non-technical domain experts to realise their vision on serious gaming.

This thesis investigates the development of a model-driven approach and technologies to aid non-technical domain experts in computer games production. It presents a novel model-driven games development framework designed to aid non-technical domain experts in producing computer games. A prototype based on the model-driven games development framework has been implemented to demonstrate the applicability of this solution. The framework has been validated through the prototypical implementations and these have been evaluated. A case study has been conducted to present a use-case scenario and to examine if this approach can help non-technical domain experts in producing computer games and also to find out if it would lower the barrier towards adoption of game-based learning as an alternative teaching and learning approach.

The work in this thesis contributes to the area of software engineering in games. The contributions made in this research includes (1) a blueprint for model-driven engineering for games development, (2) a reusable formalised approach to document computer game design and (3) a model of game software that is independent of implementation platform.

ACKNOWLEDGMENT

This doctoral study has been a journey and a life changing experience for me. I would like to take this opportunity to express my gratefulness to those who have helped and supported me throughout my doctoral study.

This thesis would not been possible if it is without the advice, guidance, support and patience from my director of study, Dr. Martin Hanneghan. His extensive knowledge in this multi-disciplinary topic of research has been a great source of expertise in driving the research forward. I am also truly indebted and very thankful to his kind help on getting me on and mentored me throughout this programme of study. I am thankful to the kind advice, guidance and motivation from my second supervisor, Professor Abdennour El Rhalibi, which I very much appreciate. I also would like to extend my sincere appreciation to my School Director, Professor Madjid Merabti for his kind advice and support. I am sincerely grateful to Dr. Christopher Carter for his kindness to share his expert knowledge in game engine and game development, and I value the advice and motivation given.

I am thankful to my faculty research administrator, Mrs. Tricia Waterson, for her patience and help on the recurring issues of the registration of my programme of study.

Above all, I would like to thank my wife, Anjoe Chou, for her personal support, patience and motivation all the times. My parents, Samuel Tang and Esther Lim for their undying love and support, without them I am would not have achieved what I have achieved today. Finally, to my beloved brother, Caleb Tang, who has given me the motivation and support me throughout this journey of mine.

Last but not least I would like to dedicate this work to my beloved son, Nathan Tang, for the strength and motivation he has given me to complete this work.

DECLARATION

I hereby declare that the work presented in this thesis is carried out by my own at Liverpool John Moores University, except for those that were acknowledged, and has not been submitted for any other degree. A list of research publications (see List of Publications) were produced during the period of this research study and some of these are presented in the context of this thesis where it deems necessary.

Stephen Tang Oon Thean (Candidate)

TABLE OF CONTENTS

LIST OF PUBLICATIONS

Journals

1. **S. Tang** and M. Hanneghan, (2011), *"State-of-the-Art Model Driven Game Development: A Survey of Technological Solutions for Game-Based Learning"*, Journal of Interactive Learning Research, 22(4), pp. 551-605. Chesapeake, VA: AACE.

2. **S. Tang**, M. Hanneghan and C. Carter, (2013), *"A Platform Independent Game Technology Model for Model Driven Serious Games Development"*, The Electronic Journal of E-Learning, Volume 11 Issue 1, pp. 61-79.

Book Chapters

3. **S. Tang** and M. Hanneghan, *(2010), "Designing Educational Games: A Pedagogical Approach,"* in Design and Implementation of Educational Games: Theoretical and Practical Perspectives, P. Zemliansky and D. Wilcox, Eds., Hershey, PA: IGI Global, ISBN13: 9781615207824, pp. 108-125.

4. **S. Tang**, M. Hanneghan and A. El Rhalibi, *(2009), "Introduction to Game-Based Learning,"* in Games-Based Learning Advancements for Multisensory Human Computer Interfaces: Techniques and Effective Practices, T. M. Connolly, M. H. Stansfield, and L. Boyle, Eds., Information Science Reference, ISBN13: 9781605663609, pp. 1-17.

Conference Papers

5. **S. Tang** and M. Hanneghan, (2013), *"A Model Driven Serious Games Development Approach for Game-based Learning"*, SERP'13 – The 2013 International Conference on Software Engineering Research and Practice, Las Vegas, USA, 22 – 25 July 2013, pp. 10-16.

6. **S. Tang**, M. Hanneghan and C. Carter, (2012), *"A Platform Independent Model for Model Driven Serious Games Development"*, in Proceedings of 6th European Conference on Games Based Learning (ECGBL2012), Cork, Ireland, 4-5 October, pp. 495 – 504.

7. **S. Tang** and M. Hanneghan, (2011), *"Game Content Model: An Ontology for Documenting Serious Game Design"*, in Proceedings of 4th International Conference on Developments in e-Systems Engineering (DESE2011), Dubai, UAE, 6-8 December, pp. 431-436.

8. **S. Tang** and M. Hanneghan, (2011), *"Fusing Games Technology and Pedagogy for Games-Based Learning Through a Model Driven Approach"*, in Proceedings of IEEE Colloquium on Humanities, Science & Engineering Research (CHUSER 2011), Penang, Malaysia, 5-6 December, pp. 380 – 385.

9. **S. Tang** and M. Hanneghan, (2010), *"A Model-Driven Framework to Support Development of Serious Games for Game-based Learning"*, in Proceedings of the 3rd International Conference on Developments in e-Systems Engineering (DESE2010), London, UK, 6-8 September, pp. 95-100.

10. **S. Tang** and M. Hanneghan**,** (2008), *"Towards a Domain Specific Modelling Language for Serious Game Design,"* in Proceedings of the 6th International Game Design and Technology Workshop (GDTW'08), Liverpool, UK, November 12-13, pp. 43-52.

11. **S. Tang** and M. Hanneghan**,** (2008), *"Game-based Learning: A Virtually-Situated Experiential Learning Approach for the 21st Century,"* in Proceedings of TARC International Conference on Learning and Teaching (TIC 2008), Kuala Lumpur, Malaysia, August 4-5.

12. **S. Tang**, M. Hanneghan, and A. El Rhalibi, (2007), *"Pedagogy Elements, Components and Structures for Serious Games Authoring Environments,"* in Proceedings of the 5th International Game Design and Technology Workshop (GDTW 2007), Liverpool, UK, pp. 26-34.

13. **S. Tang**, M. Hanneghan, and A. El Rhalibi, (2007), *"Describing Games for Learning: Terms, Scope and Learning Approaches,"* in Proceedings of the 5th International Game Design and Technology Workshop (GDTW 2007), Liverpool, UK, pp. 98-102.

14. **S. Tang**, M. Hanneghan, and A. El Rhalibi, (2006), *"Modelling Dynamic Virtual Communities within Computer Games: A Viable System Modelling (VSM) Approach,"* in Proceedings of the 4th International Game Design and Technology Workshop (GDTW'06), Liverpool, UK, November 15-16, pp. 121-125.

15. **S. Tang** and M. Hanneghan, (2005). "Educational Games Design: Model and Guidelines", in Proceedings of the 3rd International Game Design and Technology Workshop (GDTW'05), Liverpool, UK, November 8-9, pp. 84-91.

16. **S. Tang**, M. Hanneghan, and A. El Rhalibi, "Designing Challenges and Conflicts: A Tool for Structured Idea Formulation in Computer Games." Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON 2004), Het Pand, Ghent, Belgium, November 25-27, pp. 111-118.

KEY TERMS

Active Learning: A learning approach that demands active participation from students through teaching approaches that incorporate interesting learning activities to promote understanding of the concepts presented.

Computation Independent Model (CIM): A model that represents logic of system under study abstracted from the system structure.

Computer Games: Software applications that are created merely for entertainment purposes. Game software takes advantage of multimedia and other related computing technologies such as networking to enable the user (or game player) to experience goal-directed play in a rich virtual environment.

Domain Specific Modelling Language (DSML): A modelling language that is specific to a domain. Example BPMN is mainly used for modelling business process.

Educational Games: Also known as instructional games are a subset of edutainment which refers to software applications that exploit gaming technologies in creating educational content though game playing and storytelling.

Edutainment: Educational content that is presented through the integrative use of various media such as television programmes, video games, films, music, multimedia, websites and computer software to promote learning in a fun and engaging manner.

Experiential Learning: Experiential learning invites learners through direct participation in the scenario being studied rather than just studying the subject area theoretically.

Games-based learning: A learning approach derived from the use of computer games which possess educational value or other kinds of software application that use games for learning and education purposes such as learning support, teaching enhancement, assessment and evaluation of learners.

Game Content Model: The logical design specification of serious game as a model.

Game Software Model: The transformed model of the serious game specific to a technology platform.

Game Technology Model: A computation-dependent model of serious games independent of technology or language platform.

Machinima: A technique to produce pre-rendered animation using real-time game engine.

Model-Driven Engineering (MDE): A software development technique which relies on the use of models to represent aspects of software and automates the transformation of models.

Model Driven Architecture (MDA): A Model-Driven Engineering framework by Object Management Group (OMG).

Platform Independent Model (PIM): A computation dependent model of a system under study.

Platform Specific Model (PSM): A model of system under study that is specific to a technology platform.

Problem-based learning: A learning approach that invites learners to learn through solving unstructured problems.

Serious Games: A serious game is a software application designed and developed using gaming technology and principles for a primary purpose other than pure entertainment.

Situated Learning: A learning approach that require learners to be placed in a real social and physical environment that enables them to actively and experientially learn the skills and knowledge of a profession. Laboratory and workshop leaning are examples where situated learning can be used to provide "real experience" of the undertaken role and task.

Software Factory: A software development methodology that applies the principles and techniques in manufacturing to produce software with aims to reduce cost and time of development.

Training Simulators: Software systems that involve simulation of real-world experience which are intended for development of skills where challenges presented are accurately replicating a real-world scenario and require the user to overcome problems using procedural acts constrained by hardware interfaces.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 - INTRODUCTION

Game-based learning harnesses the advantages of computer games technology to create a fun, motivating and interactive virtual learning environment that promotes problem-based experiential learning. It refers to the innovative learning approach derived from the use of computer games that possess educational value or different kinds of software applications that use games for learning and education purposes such as learning support, teaching enhancement, assessment and evaluation of learners (Tang, Hanneghan, & Rhalibi, 2009). Computer games specifically designed for such purposes are generally termed as educational games or serious games which is a term used to describe computer games with embedded pedagogy (Zyda, 2005). It also encompasses games for health, advertisement, training, education, science, research, production and work, in which games technologies are used specifically for improving accessibility of simulations, modelling environments, visualisation, interfaces, delivery of messages, learning and training, and productive activities such as authoring, development or production (Sawyer & Smith, 2008). In this thesis, we use the term serious games and educational games interchangeably to refer to computer games developed for training and educational purposes.

1.1 Problem Statements

Game-based learning has been advocated by many commentators that it can provide an enhanced learning experience compared to traditional didactic methods. In recent years, great efforts have been put into realising game-based learning as demand for technology assisted learning approaches becomes popular among domain experts and academics and offer greater relevance for the so called 'PlayStation' generation of learners.

However, the adoption of such a seductive learning method engenders a range of technical, educational and pedagogical challenges, including:

- how to enable non-technical domain experts (teachers or trainers) - with little computer game development skills – to plan, develop and update their teaching material without going through endless and laborious iterative cycles of software and content development and/or adaptation;

- how to choose the right mix of entertainment and game playing to deliver the required educational and pedagogical lesson/teaching material, and;
- how to develop flexible and yet easy to use serious games development environments tailored for domain experts.

1.2 Motivation and Objective

Much research is already underway to address these stated challenges, of which some have adopted commercial off-the-shelf (COTS) games as a potential solution. However, most COTS games available are designed specifically to entertain and some even elicit violence and sexual content, thus rendering them inappropriate (but this does not imply useless) for use in an education context (Tang & Hanneghan, 2005, 2010a). Another alternative is to spearhead in-house development of serious games using open source or royalty-free game engines in collaboration with a team of developers, or 'modding' (modifying) COTS games by utilising a game editor application to create customised game objects and levels to suit the use of game-based learning. However, these approaches do not address the key challenge of facilitating the planning and development of serious games with the right mix of pedagogical, educational and fun elements.

This thesis tackles the issue on the production of computer games. It discusses the development of a model-driven development environment that can facilitate non-technical domain experts to design, develop and maintain computer games regardless of the intricacies of the game engine/environment (platform) used. By infusing games design and development with Model-Driven Engineering (MDE) practices can address the production of computer games. The model-driven game development environment captures the aspects of computer game design from users and represents it using models. Models are then analysed and transformed into working software code automatically using a built-in code generator to produce a variety of computer games. The technical aspects of games development are encapsulated from users via a user interface. Thereby allowing non-technical users to design and produce a variation of computer games quickly, easily and at less cost (in a long run).

The prototype is implemented in this research study on the Adobe Flash platform as a rich internet application (RIA) for deployment onto the web. Adobe Flash applications are supported by most browsers and, therefore, can reach out further to teachers who intend to consider game-based learning. Serious games produced using the prototype is also targeted for Adobe Flash platform to reach out to the audiences easily.

The main objective of this research study is to develop a model-driven framework that supports the development of computer games. Such a framework should provide a clear separation between artwork and technical components from the design of computer games while maintaining links to enable code generation to take place. This thesis presents the most relevant knowledge from game engineering, game design, MDE and assistive user interfaces that are essential for the development of the model-driven games development framework.

1.3 Research Contributions

The work in this thesis is unique as it draws knowledge and expertise from multiple disciplines such as game design, games development, theories of learning and model-driven engineering to address the issues stated in Section 1.1. It also makes a number of contributions to the state-of-the-art game-based learning and game development.

- **A novel model-driven games development framework to support the development of games:** This framework enables users to express design requirements of computer games through modelling and, therefore, encapsulates the technical aspects of game development. The result of this framework is a high-level tool which allows non-developer users such as teachers to produce a computer game affordably, quickly and reliably for use in game-based learning. This novel model-driven games development framework is published in (Tang & Hanneghan, 2010b) and (Tang & Hanneghan, 2011a).
- **A reusable model for representing games design:** This model improves on existing game design models and provides a formal structure for users to specify the progression of interaction and contents, properties and behaviours of in-game objects, and progression of events in scenarios. This model is a crucial part of the model-driven framework and it represents the domain-specific contents to be modelled in the game design. The work is published in (Tang & Hanneghan, 2011b) and (Tang & Hanneghan, 2011a).
- **A reusable software model for representing games independent of implementation platform:** This model represents computer games from the game software framework perspective is another part of equation of the model-driven framework. Specifications gathered through model of computer games design is mapped to this model which is then used for generation of software code. This work is published in (Tang, Hanneghan, & Carter, 2012) and (Tang, Hanneghan, & Carter, 2013).

1.4 Process and Methodology

The goal of this research study is to develop a framework which aids the development of a technological solution for non-developers user such as teachers to produce quality computer games affordably, reliably and quickly. To address this issue, the MDE approach was adopted. Aspects related to computer games design and developments are investigated. It is noted that computer games are complex applications and the quality of computer games are affected not only by the game-play, but also by artwork and technical aspects supporting the computer game such as responsiveness, quality of rendering, realism of animation and intelligence of non-player characters (NPC). A framework that separates design from assets and game functionalities, and is able to consolidate existing game technologies into a single computer games production pipeline is required.

MDE and its process are studied in detail. With the requirements in-hand, an initial draft of the model-driven framework is architected based on Object Management Group's (OMG) Model Driven Architecture (MDA) (OMG, 2001). A collection of software modelling techniques are analysed against suitability for modelling computer games. Findings from the analysis, and abstraction and relationships identified from studies related to computer games are used as guidelines for the development of a model specifically tailored for designing computer games. Both framework design and models design went through iterations of refinement and redesign to ensure the models and the framework meet all the requirements. The design of the models is continuously refined during the research study to include most aspects of computer game until it is finalised.

The front-end of the prototype, a web-based model-driven development environment, is implemented on the Adobe Flash platform with a rapid prototyping approach. The primary reason for deploying this model-driven development environment on the web is to lower the barrier of entry to this development environment so to allow practitioners of game-based learning to produce computer games using any connected desktop with a browser. The Adobe Flash platform has been chosen due to its support for the creation of rich internet applications. Various user interfaces are studied to design a suitable user interface to guide teachers in specifying design of computer games. The development began with the construction of user interfaces that capture all the required variables based on the computer game model and output the model in the form of XML. Development continued on with the implementation of a model translator in PHP that converts the design specifications to the game engine model adding in technical details that are generic to most game software frameworks and this processing is done on the server-side (back-end). PHP was chosen amongst the other server-side

scripting language for its ease of use and the support for XML parsing (Simple XML library in PHP 5.0). In addition, a code generator was implemented in PHP to generate software code for a targeted platform (and in this case the Adobe Flash Platform).

Once the prototype is implemented, a case study is carried out to evaluate the models and the framework. Details of this research study are documented in this report.

1.5 Thesis Structure

This thesis is organised into seven chapters and three appendices. Chapter 1 provides an overview of this research study. It describes the problem area and justifies the intention of this research study by clearly specifying the goals. It also highlights the contribution made from this research and the approaches taken to complete the research study.

Chapter 2 provides an overview and the essential concepts on digital games and MDE required in this thesis. It defines digital games, the game design and development process, the diversification of games design and use of game technology on non-entertainment domain and application of digital games in education. It then discusses the pros and cons of games-based learning, as well as challenges of such an approach. The chapter also introduces MDE approach and describes the MDE process. It covers key concepts such as model, domain specific modelling language, model transformation and model viewpoints, and discusses the benefits and outlines the challenges using such approach for games development.

After the fundamental concepts related to this research study are introduced, Chapter 3 surveys the literature related to model-driven game development. It investigates existing game model, game software modelling technique, game model-driven framework, game software framework, middleware, model-driven engineering tool and assistive user interface, and summarises key developments in those areas. This provides an insight on the development in the respective areas which are fundamental to this research study.

Chapter 4 proceeds to outline the requirements of a model-driven game development framework that can encapsulate technical aspects of game development from non-technical domain expert and support a myriad of game engine.

With the requirements identified, Chapter 5 presents the new model-driven game development framework designed to address the drawbacks of existing model driven game development approaches described in Chapter 3. It details the architectural strategy used in the development of the new model-driven framework, the framework itself and describes the models that are core to the

model-driven framework namely the Game Content Model (GCM), Game Technology Model (GTM) and Game Software Model (GSM) in detail. This framework is an architectural blueprint for those who intend to use this model driven approach to support development of computer games.

Once the framework is established, Chapter 6 presents a case study on application model-driven driven approach described in Chapter 5 in the domain of game-based learning. The case study shows the improvement this approach brought to the game-based learning communities in relation to production of computer games. It also critically evaluates this research work in terms of the framework proposed in Chapter 5 and the prototype presented in Appendix C.

Chapter 7 draws a conclusion to this work. It reemphasises the contributions made from this research, outlines future directions to further this research and presents some concluding remarks to this work.

Additional details that support the work presented in this thesis are made available in the appendices. Appendix A presents the Backus-Naur Form representation of Game Content Model which is useful to software engineers who intend to parse this into other form of formalised language. Appendix B shows the UML representation of the Game Technology Model which was presented in Chapter 5. Finally, Appendix C describes the prototype implementation developed specifically for practitioners of games-based learning based on the model driven game development framework presented in Chapter 5.

As a final remark, it is important to acknowledge that much of the work from this research is motivated by the vision on the future of game-based learning. It the end, it is the same vision that guides the development of the model-driven game development framework in this work. However, it is important to emphasise that the model-driven game development framework is not specific to the application on games-based learning and it can be used equally for game development in general.

CHAPTER 2 - BACKGROUND

This chapter presents an overview on computer games and its application on non-entertainment domains (Section 2.1) and provides an introduction to model-driven engineering (Section 2.2).The two main sections survey the related work and outline the fundamental concepts that are crucial in the development of a model-driven approach to assist development of computer games with an application to serious games for game-based learning.

2.1 Computer Games

Computer (video) games are interactive software applications created primarily for participatory entertainment purposes (Rollings & Adams, 2003). The terms 'computer games' and 'video games' were formerly referred to as PC-based games and console-based games but are now used interchangeably due to the blurring state of technology. Computer games as software artefacts combine multimedia and other computing technologies such as networking to enable the game player to experience goal-directed play in a virtual environment. A computer game can be represented by the three primary design schemas defined by Salen and Zimmerman (2003) in their conceptual framework as:

- Rules, which formally represent the 'mechanics' or operational constraints within the game construct, which in turn governs the level of interactivity within the game.
- Play, which represents the experiential aspect of the game and is communicated to the game player through a collection of tasks or activities that challenges the game player. The act of game-playing itself is an activity of overcoming the defined challenges and conflicts within the game using a set of permissible actions.
- Culture, refers to the beliefs and norms represented in the game world, which is often portrayed to game-players through artificial characters, objects and settings via aural and visual representation of the game world, and through storytelling.

In summary, *rules* and *culture* define the technical and intrinsic representation of some virtual "playground" to support the activity of *play*. This conceptual framework will serve as the basis to distinguish between educational games and computer games in the next section.

The focus of computer games in entertainment has always been the activity of play, which is governed by the set of formal rules defined within some cultural context. Koster (2004) defines play as a brain exercising activity that attempts to master the ability to recognize patterns in various contexts. From a pedagogical standpoint the activity of play that game-players experience is technically a loop of doing and reflecting in a motivating context that enables them to learn and hopefully master their art. Indirectly, game-players learn by doing and such an approach helps in retaining information effectively as opposed to just receiving information in a passive manner (Roussou, 2004). Such belief is centred on 'activity theory', which assumes that consciousness and activities are inseparable (Leont'ev, 1977).

2.1.1 Game Design and Development Pipelines

The design schemas described above are defined in detail by a game designer during the game design process. Game design is the creative expression of a game designer's vision of some virtual goal-directed play where the game designer engages him or herself in an activity to innovate and create a concept of play within a set of business, psychological, sociological and technological constraints. It is a process that requires one to:

i. imagine a game idea;
ii. define how the game works;
iii. document aspects of the game; and
iv. communicate such a vision and the design decisions for use as a blueprint by the development team.

This is followed by the development phase which involves a team of software engineers and artists working collaboratively to produce the interactive content as per design. The process in game development involves two different pipelines namely (1) software development pipeline and (2) digital content creation pipeline.

In the software development pipeline, software engineers will translate the game design requirements into a technical design which allows them to design software solution to support the game. Software engineers are responsible for the development of software technology that supports the game software. On a commercial scale, game software runs on a game engine – a software framework tailored to aid the production of games of the same genre. Software engineers will have to programme the mechanics game using the facilities of the game engine to ensure the game software delivers the experience defined by the game designer. To ease the process of fine-tuning the game-play

experience, software tools are usually developed by the software engineers to allow game designers to alter game variables in the game that affect the game-play. These tools are similar to the game level editor that ship with games such as the Unreal Editor. In addition to level editing tools, there are more tools which are developed in-house for use by the development team for very specific purpose such as model viewer which allows artist to check how asset would look like in the game environment.

In the digital content creation pipeline, artists will produce all the required 3d assets, 2d assets, sounds, animations and cut-scenes animation which based on the game design document. They will have to produce the assets strictly to a standard to ensure the assets can be loaded in to the game engine and rendered correctly. It is a fact that there is a wealth of knowledge and experience required in the production of computer games. This is the reason why games design and development demands expert skills and knowledge.

2.1.2 Application of Games Design and Technology to Non-Entertainment Domains

In recent years, there have been influences to apply the techniques used in game design and the game technology into other domains through *serious games*. Serious games is used to describe computer games with embedded pedagogy (Zyda, 2005). The taxonomy of serious games proposed by Sawyer and Smith (2008) (see Table 2.1) expands the scope and purpose of serious games to include games for health, advertisement, training, education, science, research, production and work, in which games technologies are used specifically for improving accessibility of simulations, modelling environments, visualisation, interfaces, delivery of messages, learning and training, and productive activities such as authoring, development or production. Some serious games featured at the Serious Games Initiative website[1] demonstrate the diverse and creative application of games technology in training and creating awareness. *Stone City* (Bogost, 2007), *Second life* (secondlife.com), *America's Army* (www.americasarmy.com), *VR Therapy for Spider Phobia* (Hoffman, Garcia-Palacios, Carlin, Furness, & Botella-Arbona, 2003) and the aforementioned *Food force* (WFPFoodForce, 2008) are some notable examples of serious games. Some serious games may not necessarily have game-play elements but can still present educational potential. *Storytelling Alice*, a programming environment designed to motivate middle school learners (especially girls) to learn computer programming through a storytelling approach, is a good example of a serious game tool for educational purposes. Although there is very little elements of game-play, it is indeed an example of game-based learning in practice.

[1] The Serious Games Initiative website can be found at http://www.seriousgames.org/

In *Storytelling Alice* learners program the animated characters to act in a story they create thereby creating the game play element themselves. Kelleher (2006) reported that learners are willing to spend 42% more time in programming using *Storytelling Alice* than the predecessor *Alice* (better known as *Generic Alice*) and devote more extra-curricular time to work on their storytelling program.

Table 2. 1: Taxonomy of serious games by Sawyer and Smith (2008).

	Games for Health	**Advergames**	**Games for Training**	**Games for Education**	**Games for Science & Research**	**Production**	**Games as Work**
Government & NGO	Public Health Education & Mass Casualty Response	Political Games	Employee Training	Inform Public	Data Collection/ Planning	Strategic & Policy Planning	Public Diplomacy, Opinion Research
Defense	Rehab & Wellness	Recruitment & Propaganda	Soldier Support Training	School House Education	War Games & Planning	War planning & weapons research	Command & Control
Healthcare	Cybertherapy/ Exergaming	Public Health Policy & Social Awareness Campaigns	Training Games for Health Professionals	Games for Patient	Visualization/ Epidemiology	Biotech manufacturing & design	Public Health Response Planning & Logistics
Marketing & Communications	Advertising Treatment	Advertising, Marketing with games, product placement	Product Use	Product Information	Opinion Research	Machinima	Opinion Research
Education	Inform about disease/ risks	Social issue games	Train teachers/ Train workforce skills	Learning	Corporate Science & Recruitment	Documentary	Teaching Distance Learning
Corporate	Employee Health Information & Wellness	Customer Education & Awareness	Employee Training	Continuing Education & Certification	Advertising/ Visualization	Strategic Planning	Command & Control
Industry	Occupational Safety	Sales & Recruitment	Employee Training	Workforce Education	Process, Optimization, Simulation	Nano/Bio-Tech Design	Command & Control

The application of gaming technologies in education has also gained tremendous interest from different sectors including government, academia and industry (BECTa, 2001, 2006; FAS, 2006a). Many agree that it is now appropriate to take advantage of gaming technologies to create a new generation of educational technology tools to equip learners of all ages with necessary skills through experiential learning (FAS, 2006a). As positive impressions on digital games continue to spread, various programmes of research have been conducted to realise the use of game content at various levels of learning.

In general, game-based learning refers to the innovative learning approach derived from the use of computer games that possess educational value or different kinds of software applications that use games for learning and education purposes such as learning support, teaching enhancement, assessment and evaluation of learners. The term 'game-based learning' can also refer to the use of

non-digital games such as card games (Baker, Navarro, & Hoek, 2005) and casino chips (E. Cook & Hazelwood, 2002). It is an activity to engage and hold learners in focus by encouraging them to participate during the lesson through game-play. More specific terms that refer to the use of computer games in learning and education include 'digital game-based learning', which was coined by Prensky (2001), and 'games-based eLearning' by Connolly and Stansfield (2007).

In game-based learning environments learners are presented with learning material in the form of narrative and storytelling and they learn through game-playing and studying the properties and behaviour of in-game components, the relationship between these in-game components and the solving of problems in the defined scenario (Tang, et al., 2009). From a learning theory perspective, game-based learning possesses characteristics such as:

- motivating and engaging;
- requires participation from learners;
- has clear learning objectives defined in the game-play and scenarios presented while knowledge can be imparted through storytelling and narrative;
- scenarios defined are reflective and transferable to the real-world experience;
- provides freedom to interact in the game world through a set of defined actions;
- provides clearly defined feedback for every action taken;
- both assessment and lesson can take place during game-play;
- matches learner's pace and intellectual ability;
- highly scalable so can be used for educating large numbers of learners concurrently.

Many researchers believe that such innovation in learning technology can better motivate present day entertainment-driven learners to experience learning through meaningful activities defined in the game context as opposed to traditional pedagogic approaches. Findings from initial research studies also show that computer games can be used to acquire certain cognitive abilities and improve their understanding in topics presented (Aguilera & Mendiz, 2003; BECTa, 2006; Jenkins, Klopfer, Squire, & Tan, 2003). These preliminary results are convincing and have gained tremendous interest from different sectors including government, academia and industry to further explore the benefits of such opportunity (BECTa, 2006; FAS, 2006b). Many also agree that it is now appropriate to take advantage of gaming technologies to create a new generation of educational technology tools to equip learners of all ages with necessary skills through experiential learning (FAS, 2006b).

Computer games for use in game-based learning are generally termed 'educational games'. Computer games and educational games share many common technical features but differ in their intended use and design of content. Computer games are primarily designed for entertainment purposes while educational games are intended to impart knowledge or skills development (although some educational aspects and entertainment aspects exist in both fields). Therefore the real distinction between computer games and educational games can only be further explained through the definition of the design schemas *play, rules* and *culture* based on the purpose defined.

Play in the context of computer games has always been perceived as an activity of enjoyment or recreation instead of serious or practical purpose. Contrary to computer games, *play* in the context of educational games should be defined as meaningful learning activities that promote the formation of new concepts and development of cognitive skills. These meaningful learning activities are interactions designed with an aim to educate learners through the principle of cause and effect. *Rules* and *culture* have to accommodate the direction of play defined for either purpose; entertainment or learning, or both. In fact, many well-designed computer games are indeed educational although they are lacking in the integration of knowledge and training in skills that are considered educational.

Rules that govern game-play in educational games are coupled with measureable learning objectives that are assessable via interactivity. Although computer games have similar measurable objectives, game objectives are designed to steer game-play towards entertaining play and may not be applicable in reality. Grand Theft Auto IV (Rockstar North, 2008), a controversial but a very successful[2] computer game where the game-player takes on the role of criminal in a big city and participates in occasional criminal activities such as occasional taxi driving, fire-fighting, pimping, street racing and other regular crime features such as bank robbery and assassination, is an example of rules of game-play that contradicts with society norms. These rules can also be in the form of distinct challenges that place demands on the learner to solve a variety of problems cognitively and possibly requiring hand-eye coordination or manual dexterity. These challenges also exist in computer games but may be presented in a fictitious context that is irrelevant to any real-world analogy and may lack accuracy in representation. Mechanistic rules underlying game-play for educational games can range from simplistic to extremely complex representation (for example on a par with a true simulator) depending on the subject matter.

[2]Based on total sales.

The details of *culture* in educational games depend largely on the subject matter and the designed learning objectives. Ideally educational games should exhibit belief and norms from some real-world scenario to facilitate knowledge transfer from the game world to reality. However most educational games have the world set in a fantasy environment as this may increase the intrinsic motivation of learners according to Malone & Lepper (1987). In such a context, belief and norms should be defined according to the learning objectives and reflect some degree of truthfulness and relation to the real-world to sustain the educational values that distinguish educational games from computer games. Story and narrative are often used to set the scene and immerse game-players into the game world both in computer games and educational games from various perspectives. The difference, however, lies in the defined events (game-play sessions) driven by the story whether the events introduced in the game-play are some form of meaningful activities that would help game-players to understand the subject or a playground merely for eliciting fun. In fact it is more natural to use dialogue as a method of storytelling and information dissemination to the game-player via artificial characters instead of narration. Other forms of content beside narration, such as the visual element, need not be ultra-realistic although it is desirable (but costly) to include such a requirement in educational games. Visual elements in the form of avatars and objects are sufficient for the purpose of learning. Table 2.2 below summarises the main differences between computer games and educational games in relation to play, rules and culture.

Table 2. 2: Differences between computer games and educational games in purpose, play, rules and culture.

	Computer games as Entertainment software	**Computer games as Edutainment software**
Purpose	For entertainment purposes. Context presented is mostly fictitious or fantasy based.	For learning and skills development purposes. May be a form of entertainment based on the interpretation of the learner.
Rules	Interactions designed primarily for entertainment purposes with directed objectives that can be driven by storytelling. Interactions resemble the real-world interaction in a simplified or abstract approach.	Interaction designed for learning purposes with meaningful responses and measurable outcomes. Knowledge is disseminated through events triggered by specially designed interactions and dialogue.
Play	Rules are designed to accommodate the activity of play, which are often tuned for playability rather than reflecting the real-world.	Rules are designed for specific learning outcomes that can be used to measure the interactions during "serious play". Rules can be simplified or made complex to support the activity of play.
Aesthetic representation	Beliefs, norms and world setting presented visually and via narrative often set in an imaginary world that is represented artistically and often exaggerated.	Beliefs, norms and world setting presented visually and via narrative that are related to knowledge domain, reflect truthfulness and have direct and explicit relation to real-world events. Game world maybe set in an imaginary world.

Some examples of educational games are *Food Force* and *Hot Shot Business*. *Food Force* is an educational game published by the United Nations World Food Programme (WFP) to educate children between the ages of 8 – 13 about the fight against world hunger. Set in a fictitious island called Sheylan in the Indian Ocean players are taken through six different missions with specific learning objectives: (1) *Air Surveillance* - The causes of hunger and malnutrition; (2) *Energy Pacs* - Nutrition and the cost of feeding the hungry; (3) *Airdrop* - WFP's emergency response; (4) *Locate and Dispatch* - Global food procurement; (5) *The Food Run* - Land-based logistics; and (6) *Future Farming* - Long-term food aid projects. Simple game-play is introduced in each mission, for example, in the *Energy Pacs* mission game-players are required to purchase food items such as rice, beans, vegetable oil, sugar and iodised salt with the budget of USD0.30 per person per meal within 2 minutes.

Disney's *Hot Shot Business*[3] is a business simulation game designed for children between the ages of 9-12 to help them learn the required skills to become a successful business owner (Everett, 2003). The educational game is designed collaboratively with the Ewing Marion Kauffman Foundation, which funds education and entrepreneurship, to support its corporate curriculum. In *Hot Shot Business* game-players can choose to own various types of business such as a candy factory, pet spa, landscaping service, comic shop, skateboard factory, magic shop and travel agent. As the game progresses game-players are required to set-up the business, sell services or products, respond to market needs, price the services and products accordingly, market the services or products and compete with other competitors. Business strategies are then evaluated to reflect the game-players' performance.

Other examples of educational games designed for learners in middle school, institutions of higher learning and adults include:

- *UNIGAME* – a web-based game that encourages learners in higher education institutions to search for information, discuss topics and arrive at a consensus using a problem solving approach (Pivec & Dziabenko, 2004).
- *CyberCIEGE* – a computer game that teaches information assurance concepts through the simulation of an IT firm where learners take on the role of decision maker to satisfy the needs of virtual users while also protecting valuable information assets from cyber-criminals (Irvine, Thompson, & Allen, 2005).

[3] Hot Shot Business can be played online at www.hotshotbusiness.com

- *Supercharged!* – a computer game in which learners in middle school can learn about electrostatic properties by navigating through an electrostatic maze controlling the charge of the spaceship through careful placement of charged particles (Squire, Barnett, Grant, & Higginbotham, 2004).

There is currently a larger focus on educational games for children than any other age group. This is mainly due to the early stage of edutainment research that focuses mainly on child education. In addition, many educational games are developed quicker and more cost effectively for young children since they have a lower expectation of the sophistication of the interactive content compared to teenagers and adults. Training simulators are more popular among adults especially in the field of aviation (Telfer, 1993) and medicine (Colt, Crawford, & Galbraith, 2001).

2.1.3 Technologies Supporting Game Creation

In recent years, games technologies are becoming more mainstream and accessible by the public. Technically-able domain experts can now choose to develop their own game software using either royalty free game engines or open source game engines. These are industry standard framework that allows one to develop complex game software and will require experience, knowledge and skill on software development. Many would find it technically challenging and they would want to acquire experts help or even outsource the game creation to game developers at a cost.

There has also been advancement in the tools that support game developments. In the past, tools are propriety technologies developed for in-house usage to aid game developers in games development with the aim to increase productivity. Most of these tools are developed to for a specific purpose and often it has poor user interfaces which are not suitable for non-developers users. However, this has changed. There are now readily accessible tools for the domain experts. Tools such as the Unreal Development Kit[4] and the Unity Game Engine[5] provide users with a collection of tools with improved user interfaces to help make game development easier. These tools are now used widely by the game hobbyist community and indie game developers to develop games either as a hobby or for commercial purpose. Although tools can help to simplify some aspects of game development such as providing better visualisation of the game and loading of asset into game engine, users will need to have the knowledge and skills to be able to develop a game. For the less-technical domain experts, they can explore into the use of game creation tools that does not requires any form

[4] www.unrealengine.com/udk/

[5] www.unity3d.com

of programming such as Scratch (Maloney et al., 2004) and more recently released Game Salad[6]. These tools provide the user interfaces that replace the manual programming of codes and the facility to reuse the built-in features. Although these tools are made more user-friendly for the less technical users, it will still require users to have some knowledge of game creation to be able create a game.

Another option to create a game with the use of tool is through the process of "modding" or modifying an existing game. Some games such as Unreal Tournament III (2007) ships with a game editor tool which allows users to create their own game level by reusing the game functionality and assets. Computer-savvy domain experts would be able to take advantage of this facility to repurpose a commercial off-the-shelf game for their intended purpose. Although the game editor tool has an improved user interface compared to in-house development tools, it will still require domain experts to be familiar with the game before they can begin to modify the game. "Modding" a game does not provide much flexibility to domain experts who plan to develop their own custom game because it only allows permit customisation of content within the scope of the game mechanics and technology supported.

2.1.4 Challenge for the Non-Technical Domain Experts

The idea of applying games design and games technology to non-entertainment domains is interesting and it attracts attention of many. There are now a growing number of domain experts who which to develop their own games to communicate their ideas, to educate a group of learner, to promote a product and to promote health and well-being through the serious game. However, as we have identified in Section 1.1 and Section 2.1.3, there is not a single tool that is simple enough made available for the non-technical domain experts to produce serious games for use in their respective domains. Most of the tools available out there are designed specifically for in-house development purpose and will require substantial experience and knowledge to use. This means that the non-technical domain experts would have to rely on the experts in the production of serious games and this may hinder many from using the serious games as a medium to engage with their audiences. Thus, we believe there is a need to investigate the use of model-driven software engineering approaches to facilitate non-technical domain experts to plan, develop and maintain computer games regardless of the intricacies of the game engine/environment (platform) used. In the next section, we introduce model-driven engineering and highlight important concepts that are used throughout this thesis.

[6] http://gamesalad.com/

2.2 Model-Driven Engineering

Model-Driven Engineering (MDE) refers to a software development approach that relies extensively on the use of graphical or logical models to represent aspects of software and automates the transformation of models into more refined software artefacts. Models are primary artefacts in the development process expressed using a *Domain Specific Modelling Language* (DSML) to formally represent the structure, behaviour and requirements of a particular domain. Aspects of models are analysed and translated through transformation engines and generators to synthesize software artefacts consistent with these models. This approach can help in alleviating the platform complexity and expressing domain concepts effectively (Schmidt, 2006). Another term synonymous to MDE is Model Driven Development (MDD). In this thesis, MDE and MDD are used interchangeably.

MDE is generally domain specific and only suitable for developing software specific to a targeted platform with the aim of automating manual coding. Expertise of programmers is embedded into the underlying framework and code generators allowing domain experts to provide complete specifications of the problem domain without worrying about the technical aspects of software development (S. Kelly & Tolvanen, 2008).

Over the years researchers have striven to improve MDE technologies applying lessons learnt from earlier efforts in developing platforms and languages with high-levels of abstraction. In recent years, researchers have opted to adapt the MDE approach deviating away from traditional software development approaches with the intent to separate the model from any platform dependent implementation. MDE approaches have been applied in areas such as web application modelling (Schauerhuber, Wimmer, & Kapsammer, 2006), web services modelling (Grønmo, Skogan, Solheim, & Oldevik, 2004), security modelling for distributed systems (Lodderstedt, Basin, & Doser, 2002), QoS validation of component-based software systems (Hill, Tambe, & Gokhale, 2007), control and automation systems modelling (Thramboulidis, 2004), user interface modelling (Sottet et al., 2006), business software modelling (Hildenbrand & Korthaus, 2004), content repurposing (Obrenovic, Starcevic, & Selic, 2004) and context-awareness web application modelling (Ceri, Daniel, Facca, & Matera, 2007). Current research in MDE focuses mostly on systematic reuse of the development experience through the concept of the software factory, systematic software testing and compilation technologies (France & Rumpe, 2007).

2.2.1 MDE Process

The MDE process can be represented with two distinct approaches; *top-down* and *bottom-up*. Both approaches are driven by a set of different motivations. Justified by the need to generate software code to a variety of targeted platforms, the top-down approach examines variations from the conceptual level. This approach can result in the creation of better designed, robust and consistent models. Defining a complex model may seem a straight forward task but implementing such a model can be daunting (S. Cook, Jones, Kent, & Wills, 2007). The top-down approach can be defined by the following succession of activities (Fondement & Silaghi, 2004):

I. Identifying the level of abstraction and platforms to be integrated;
II. Specifying modelling notation and abstract syntax to be used at each level of abstraction;
III. Specifying refinement processes, and platform and related information to be integrated in lower level of abstraction;
IV. Defining generators for modelling language used at the lowest level of abstractions (and even deployment of such code);
V. Specifying verifier and validator to check against the upper level model, and generation of test cases for system under development.

In contrast, the bottom-up approach addresses software variability through observing variation in existing codes. It is a client-oriented approach that requires more upfront effort, is time consuming, and may limit the extension of models in the future, but is simpler to implement. The bottom-up MDE process is defined as follows (S. Cook, et al., 2007):

I. Identify variability from business concepts;
II. Conduct variability analysis on the existing code;
III. Define a domain model;
IV. Specify modelling notation for the DSML;
V. Refine the DSML;
VI. Define generator based on existing code;

Cook et. al. also (2007) advices that it is best to alternate between the two approaches and work incrementally to ensure models and transformations are consistent.

2.2.2 Model

At the core of MDE is the model. A model is defined as "a simplification of a system built with an intended goal in mind [that] should be able to answer questions in place of the actual system" (Bézivin & Gerbé, 2001). Technically a model is described as a set of statements that effectively describes a *system-under-study* (SUS) (Seidewitz, 2003). Effectively, a model is a graphical or logical representation of a SUS (Favre & Nguyen, 2005). In a software context, models are used as a form of conceptual representation of a software system to facilitate the production of concrete software (Bézivin, 2004). Alternatively, it can also be used as a specification for beginners to study a system. In a model-driven approach, the model is used to assist the system modeller to more accurately describe the SUS. The model is expressed by the system modeller through statements, which are expressions that hold truths about the SUS. The validity of the model of the SUS is dependent on the correctness and completeness of statements describing it. Statements can be expressed informally through natural language, but are best constructed using formal notations which adhere to the grammar of the formal language. Such formal languages are generally termed as modelling languages or Domain-Specific Modelling Language (DSML).

2.2.3 Domain-Specific Modelling Language (DSML)

There are many modelling languages available for use in modelling software including Unified Modelling Language (UML), Integration Definition for Function Modelling (IDEF), Z-notation, flowcharts, Data-Flow Diagrams (DFD), Petri-nets and State Diagrams for example. Experienced modellers can model any type of SUS using any of the aforementioned generic software modelling language as a Platform Independent Model (PIM) or concretely as Platform Specific Model (PSM) (Kent, 2002). Generic software modelling languages provide a high degree of flexibility and have no domain-specific rules integrated. However, this may hinder non-technical domain experts when expressing a SUS within the constraint of the modelling language which are not tailored for their domain (Tolvanen, 2006). Even all-purpose modelling languages such as UML are reported to be complex and practically limit their application of into MDD environments (France, Ghosh, Dinh-Trong, & Solberg, 2006).

A DSML is designed strictly to embed concepts, language and associated rules of a specific domain to better handle complexity of that domain, easing the modelling task for domain experts, and facilitate automatic generation of full software code independent of platform and/or representation of the SUS (S. Kelly & Tolvanen, 2008). A DSML can be developed to suit any domain, for example, in

education - modelling lessons (Koper & Manderveld, 2004); in ecology – simulating landscape dynamics (Fall & Fall, 2001); in electronic engineering – analysing energy estimation (Choi, Jang, Mohanty, & Prasanna, 2003); in software engineering – integrating large-scale systems(Shetty et al., 2005) and developing test plans (Katara, Kervinen, Maunumaa, Paakkonen, & Satama, 2006). In the above examples, some are intended for capturing only the specification of a domain, while others are for automating creation of software and/or other artefacts.

The process of defining a DSML shares many similarities with the MDE process described in Section 2.2.1 and is summarised in four steps as follows (Hammond, 2007):

I. Identifying abstractions and relationships;
II. Specifying the language concepts and associated rules (meta-model);
III. Creating the visual representation of the language (notation);
IV. Defining the generators for model checking, code, documentation, etc.

2.2.4 Model Viewpoint

The aim of MDE is to raise abstraction to a higher level while suppressing irrelevant details. Through abstraction, a model is created to reflect certain criteria which determine what should be included. Different abstraction criteria can provide different viewpoints of the SUS. The basis of MDE is the domain viewpoint which represents the problem domain ontologically. This model can then be replaced with code directly using a code generator tailored specifically for the model. Intermediary models can also be introduced to increase the interoperability of the SUS. The software model is a typical example of intermediary model that represents the SUS programmatically.

Although the goal of MDE is to enable a SUS to be translated from software model to a complete system, more intermediary models may be introduced to refine the software model to a targeted platform to maximise interoperability of systems. The Object Management Group's (OMG) Model Driven Architecture (MDA) is one such example. MDA is a framework for MDE standards that advocates modelling systems using UML to separating business or application logic from platform implementation in order to achieve maximum interoperability of systems (Kleppe, Warmer, & Bast, 2003; Poole, 2001). MDA consists of three viewpoints (Gaševic, Djuric, & Devedžic, 2006; OMG, 2003) namely:

- Computation Independent Model (CIM) – A model that represents the logic of the SUS abstracted from the system structure. It only provides an ontological description of the system from a domain perspective.

- Platform Independent Model (PIM) – A computation-dependent model that is not tied to any technology or language platform. It serves as the core development artefacts of the systems.
- Platform Specific Model (PSM) – A model of software specific to a technology platform but yet to be represented in a real implementation (software code).

There are no limits on the maximum number of viewpoints that should be included in a model-driven framework. A higher number of viewpoints representing different levels of abstraction will increase reusability of models and interoperability of systems. However, this also implies that more refinement needs to be done to each subsequent model before it can be translated into software codes.

2.2.5 Model Transformation

Models can be translated into a new target model throughout the pipeline of MDE and finally to a complete implementation of a software system (or documentation of a complete software system). The relationship between the source model and the target model is termed as model transformation (France & Rumpe, 2007). The transformation process is a unidirectional process (S. Kelly & Tolvanen, 2008) which can be done either manually or automated using MDE tools.

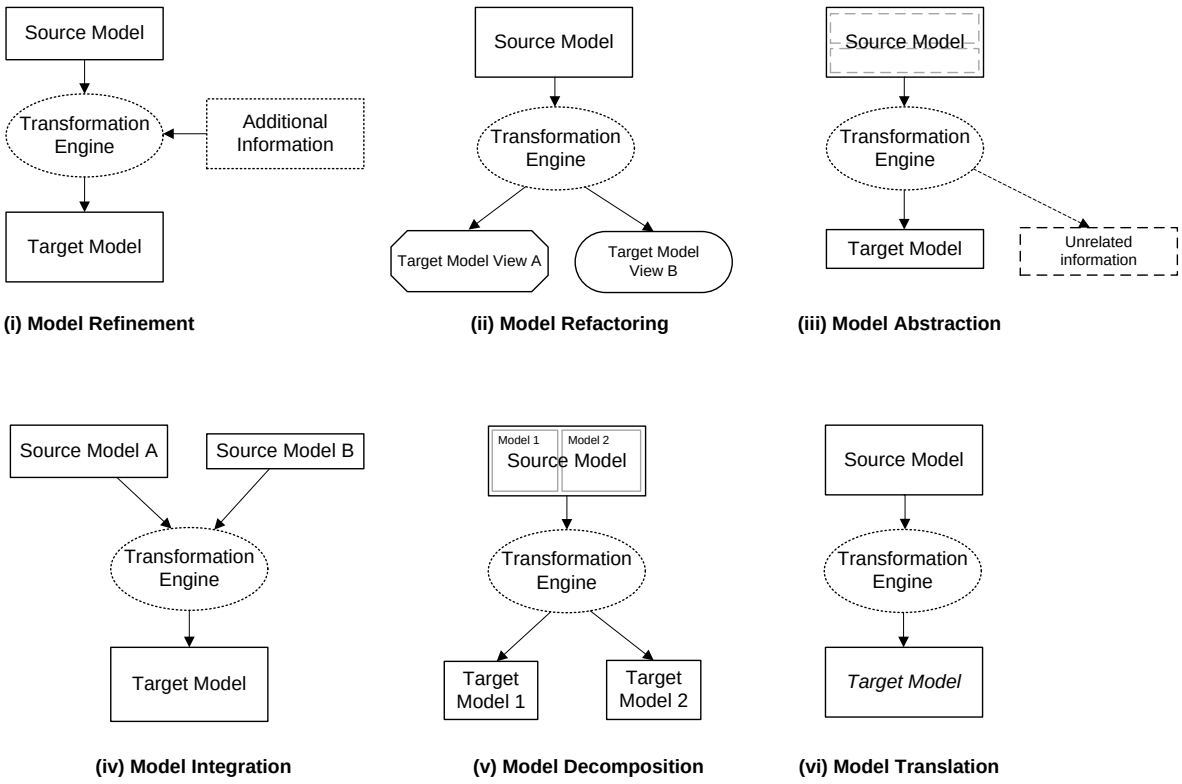

Figure 2.1: Model-to-Model Transformations

Transformation can either be a *model-to-model* transformation or *model-to-artefact* transformation. In a *model-to-model* transformation, the form of transformation includes: (i) *model refinement* where a source model is added with additional information to form a target model; (ii) *model refactoring* where a source model produces multiple target models with different viewpoints; (iii) *model abstraction* where a source model has certain information discarded to produce a target model with higher abstraction; (iv) *model integration* where source models with different viewpoints are integrated to create a target model with a combined viewpoint; (v) *model decomposition* where a single model is decomposed into multiple target models; and (vi) *model translation* where the source model is expressed in different languages (France & Rumpe, 2007). Such processes can be automated using MDE tools generically known as *transformation engines*. As for the *model-to-artefact* transformation, transformation can be in form of *code generation* where the model is translated directly into a software artefact such as scripts, configurations, codes or middleware languages accepted by the targeted platform, or *documentation generation* where specifications are represented in the form of textual or graphical output such as UML diagrams. Generators are MDE tools that are suitable for such transformations.

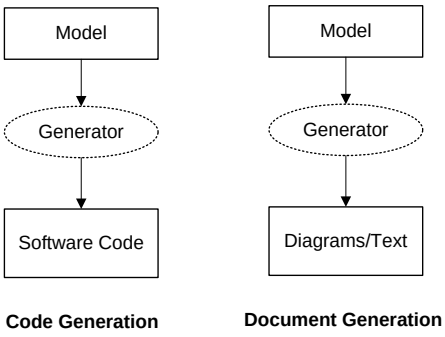

Figure 2.2: Model-to-Code Transformations

The number of model-to-model transformations required for an end-to-end transformation depends on the level of abstraction in the model driven framework. One-tier architectures can directly transform a model into code, but multiple-tier architectures such as the Object Management Group's Model Driven Architecture (MDA) would have to undergo two cycles of model-to-model transformations before it can perform model-to-artefact transformation.

2.2.6 Roles in MDE

Typically, software engineers are responsible in gathering the requirement specification of a SUS from domain users. In MDE, such roles have changed. There are three main roles in MDE (S. Kelly & Tolvanen, 2008):

- ***Domain experts***– These are people who know the terms, concepts and rules of the domain. Their knowledge in the area helps developer to formalise it into the MDE framework.
- ***Domain users***– These are people who model the SUS using a modelling language to generate applications. They can be anyone in the domain who wants to take advantage of software generation technology to improve productivity.
- ***Developers***– This is the team that contribute the most to the realisation of MDE. They are involved in developing the modelling languages, the domain framework and the tools to encapsulate the technical aspects of modelling through User Interfaces (UI) and to automate the transformation of models. Usability engineers may also be part of the development team to ensure usability of the modelling language.

Although there is a change in roles, personnel involved in a software development project remain unchanged. In MDE, human resource is optimised. Once an MDD environment is in place, domain users can generate applications through modelling without much assistance from domain experts or developers. The tasks for domain experts and developers are then to maintain the framework and proprietary tools whenever there are changes to the framework.

2.2.7 Advantages and Disadvantages

Some of the notable benefits of MDE from a software development context include an increase in productivity, promotion of interoperability and portability among different technology platforms and support for generation of documents to support software maintenance (Kleppe, et al., 2003). In addition, MDE can also lead to better quality applications due to the integration of domain rules into the modelling language which minimises modelling error and promotes reliable mapping from model to code (S. Kelly & Tolvanen, 2008). For an obvious advantage, MDE encapsulates the technical aspects of development and, therefore, provides the necessary aid to domain users to develop applications with minimal low-level technical knowledge.

The downside to MDE is the high cost of development which may or may not be justifiable with the amount of turnover a project can generate. Overall, MDE can offer great benefits for businesses

and for non-technical domain users as it opens the door to create quality, reliable applications easily and affordably.

2.2.8 Challenges

The benefits of MDE are certainly attractive to many, but realisation of an MDE solution is still challenging for developers. France & Rumpe (2007) classified the challenges of MDE into three categories as follows:

- **Modelling Language Challenges:** When defining a modelling language, language developers are challenged to create the problem-level abstractions and formalise the semantics of the language. Creating the problem-level abstractions requires the language developer to fully understand the concepts, terminologies and relationships that represent the problem domain. This task can be made easier with the help of domain experts. While creating a notation for the language, iteration of user testing will help to ensure the notation is acceptable and usable.
- **Separation of Concerns Challenges:** In complex software, problems may arise when modelling involves multiple and overlapping views using heterogeneous languages. Some features of the system may have conflicts due to the design and it is a real challenge for developers to make sure that features modelled can interact with each other. Separation of these design concerns is necessary to avoid costly system failure and can be achieved through a well-defined viewpoint. Undertaking such a task will require experience and involve several cycles of design-test-evaluate before coherent model design can be achieved.
- **Model Manipulation and Management Concerns:** More challenges await developers in matters pertaining to transformation of models. These include maintaining the consistency of models transformed, testing of transformation and integration of generated code with foreign code. The transformation process also adds complexity to management of models when revisions are made to models in terms of versioning, repository storage of previous models and maintaining integrity of relationships of models.

These challenges are greater for complex software such as educational games. In addition to functional aspects of educational games, the MDE developers will have to address the incorporation of media such as 2D graphics, 3D models, sound and animation data, and the extensibility of game functionalities such as game physics and game artificial intelligence (AI) while assuring educational

games produced are of acceptable quality. The MDE approach may also limit innovation of educational games due to the constraints of customisation. Although this is a trade-off for the MDE approach, the benefits listed in Section 2.2.7 far outweigh it.

2.3 Chapter Summary

In this chapter we have introduced computer games and the process involves in designing and development of game. We have also described its potential in the non-entertainment domains particularly in the area of education or game-based learning. Our analysis and review of the literature indicates that game-based learning has a learning model embedded within a software environment and it shares many of characteristics from learning approaches such as active learning, experiential learning and situated learning. Furthermore game-based learning is able to address the various learner styles through the experiential learning approach in game playing and can fill the spare time of young adults with productive play rather than entertaining play to better prepare the workforce. As we have highlighted earlier in Chapter 1, there is a need to provide a technological aid to the non-technical domain experts who wish to develop their own serious games for their respective domains. Our review of the literature on MDE highlights great benefits of such practice in software engineering. MDE techniques can help to formally gather the necessary requirements of a serious game through formalised language in the form of textual or visual notations, and later be translated into working source code specific to a game technology platform. Such an approach encapsulates the technical aspects of serious games development and therefore provides the necessary aid to non-technical domain experts to produce their serious games with minimal low-level technical knowledge in game development. But adoption of such an approach in development of serious games will require one to overcome challenges in defining the modelling language, separating modelling into multiple viewpoints, and manipulation and management of models to artefacts. These challenges will be addressed in the remainder of this thesis.

CHAPTER 3 - STATE OF THE ART IN MODEL-DRIVEN DEVELOPMENT FOR COMPUTER GAMES

Games design and development is a creative and innovative field which demands specialist knowledge in the production of entertaining interactive content. In recent years, the application of games design and technologies has been expanded to other areas such as communication, marketing, training and education under the serious games initiatives. There are a growing number of users keen to 'gamify' their digital content and introduce gaming experience to their industry. The public are ready for such an approach as they are increasingly familiar with gaming culture (Pearce, 2006) and the technology to support game-based learning is more affordable and widely available. The only puzzle to realise this vision is the availability of high-level tools made specifically for non-technical domain experts to assist them in creating computer games in an affordable and timely manner. As highlighted earlier in this thesis, MDE offers the basis for development of high-level tools. Development of a model-driven games development environment will require a game modelling language, a framework that can integrate media and game functionalities, transformation engines and code generators. In this chapter, we survey the literature in the domain of game design, game software engineering and MDE, and summarise the key developments that are propelling games development towards the model driven approach.

3.1 Design Languages for Games

Design is generally defined as "the human capacity to shape and make our environment in ways without precedent in nature, to serve our needs and give meaning to our lives" (Heskett, 2005). From a game design perspective, it is regarded as the creative expression of a game designer's vision of some virtual goal-directed play where the game designer engages him or herself in an activity to innovate and create a concept of play within a set of business, psychological, sociological and technological constraints. This activity demands documentation to record the decisions of choices which serve as a blueprint to communicate the underlying vision to the development team.

Conventionally, game design is documented as a collection of game design documents in the form of high-concept document, treatment document and detailed design document. This approach is still taught through texts (see for example Rouse (2001), Rollings and Adams (2006; 2003), Bates (2004) and Fullerton (2008)). Documenting the aspects of game design is a somewhat informal process driven by creativity (Crawford, 1982; Fullerton, 2008). Design within a specific game genre is also bound by rules and certain expectations. For example, when designing a role-playing game (RPG) such as the Final Fantasy[7] series, emphasis should be given to the design of a customisable player-character that can be improved throughout game-play and a story built around a quest that can engage game players in an explorative world to satisfy the appetite for adventure. In contrast to this, construction and management games such as the SimCity[8] series have their design focused on the economy system and the construction process. At present, samples of game design documents and templates for game design from game design references (Fullerton, 2008; Rollings & Adams, 2003; Rouse, 2001) are the only available guide for documenting game design.

In recent years, various studies (Adams, 2004; El-Rhalibi, Hanneghan, Tang, & England, 2005; Onder, 2002; Tang, Hanneghan, & El-Rhalibi, 2004; Taylor, Baskett, Hughes, & Wade, 2007; Taylor, Gresty, & Baskett, 2006) have proposed the use of modelling languages to help game designers systemically and systematically design computer games. Our definition of an ideal game design language would consist of game-related vocabularies and game-concept related grammars that allow game designers to formally express their game idea.

The *Soft Systems Methodology* (SSM) (Checkland & Scholes, 1990) is an approach based on the theory of systemic thinking to study the components of a system and their relationships with one another in order to develop an understanding about that system. This could be used as a formalised method for designing games. In the context of games design, SSM can assist game designers in designing game flow and identifying appropriate challenges and conflicts (El-Rhalibi, et al., 2005; Tang, et al., 2004; Taylor, et al., 2007). A game idea is elaborated in the form of a *Rich Picture*™ where purposeful activity systems are identified and a relevant *Root Definition* (RD) is defined based on *Customers, Actors, Transformation process, Weltanschauung (or world view), Owners* and *Environmental constraint* (CATWOE) analysis. An example of this is illustrated in Figure 3.1. SSM offers a methodological and systemic approach to game design through an iterative process of logical

[7] Read more about Final Fantasy from http://www.playonline.com/ff11us/index.shtml.
[8] More details about SimCity can be found at http://simcitysocieties.ea.com/index.php.

reasoning, but it lacks aspects that describe the story, user interfaces, input control, objectives and game-play which are essential for creating compelling educational games.

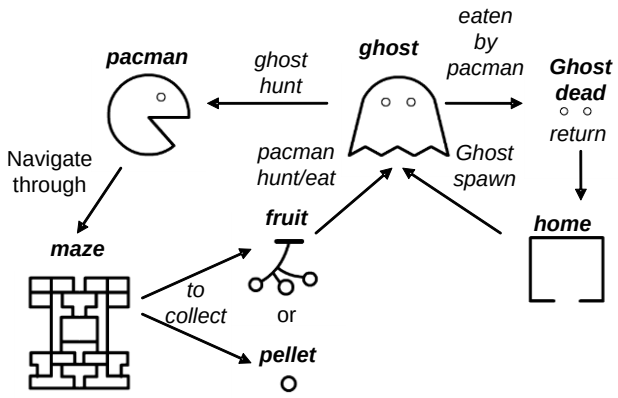

Figure 3.1: Pacman Game represented in Rich Pictures (Tang, et al., 2004)

The *story beat diagram* proposed by Onder (2002) is composed of ovals and arrows that represent the progression of a story (see Figure 3.2). It illustrates the possible routes for the game player to explore within the defined interactive story. In addition to the story beat diagram, further details about objects and cast members, locations of scenes, setup of scenes, player interactions and associated responses, and distribution of game-play within the storyline are required in writing to complete the specification of a game design. Essentially, the story beat diagram can only capture the flow of game-play ordered within a story while the rest of the requirements are recorded separately.

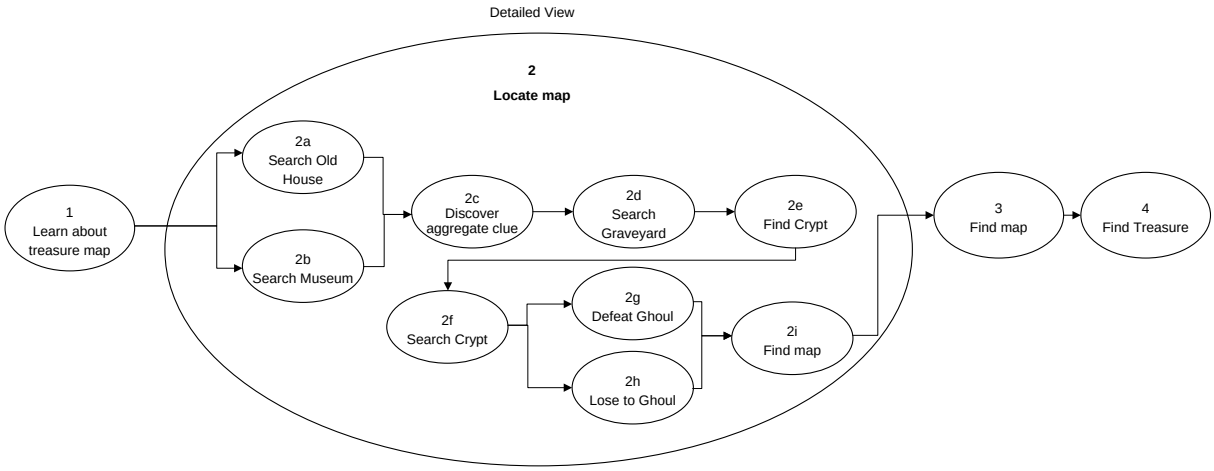

Figure 3.2: Example of Story Beat Diagram (Onder, 2002)

The *flowboard* is an adaptation of the storyboard and flowchart to document game structure (Adams, 2004). It uses rectangles to denote segments of a game such as splash screen, menu screen,

result screen, game screen, high-score tables and other, while arrows with conditions denote the constraints or condition that allow progression from one segment to another (see example in Figure 3.3). The flowboard is a simple and useful tool to model the various modes offered in a game, but it lacks the ability to record essential game design details required to complete the design of a game such as objects and character's behaviour, challenges, conflicts, objectives, environment, user interfaces, storytelling and narration.

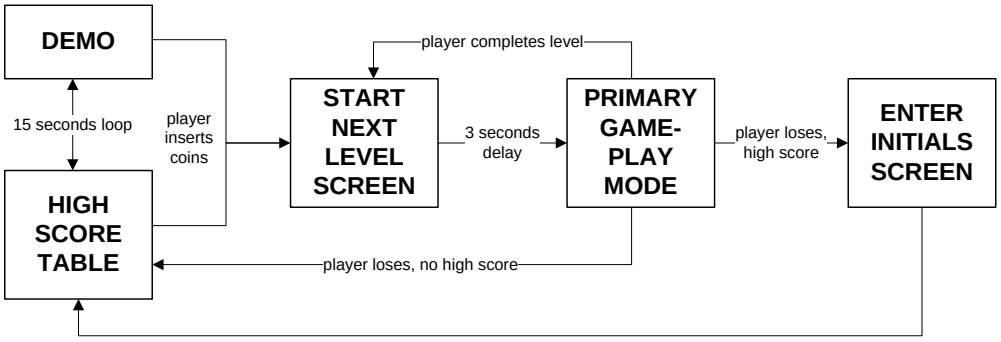

Figure 3.3: Example of Flowboard (Adams, 2004)

There have also been efforts to use software modelling languages such as *the Unified Modelling Language (UML)* for game design. The *Computer Game Flow Design Diagram* proposed by Taylor, Gresty, & Baskett (2006) adapts the language from the UML use-case diagram to form a new diagramming language to document game flow. The computer game flow diagram uses arrows to denote the direction of game-flow, rectangles for fixed game objects, ovals for mobile game objects (game objects that move around) and pseudo code to describe interactions and responses (see example in Figure 3.4). In addition to modelling linear game flow, it can also be used to model non-linear flow where games can be designed to provide game players greater freedom to navigate within the game world achieving game objectives in a non-ordered fashion. The use of symbols and labelling of objects, characters and environments are clear, but the method is insufficient for describing details of objects, characters, objectives, environments, narration, storyline and user interface. Although the use of pseudo code can help to describe many more details programmatically, this approach is too technically demanding for non-technical domain experts to document design details adequately.

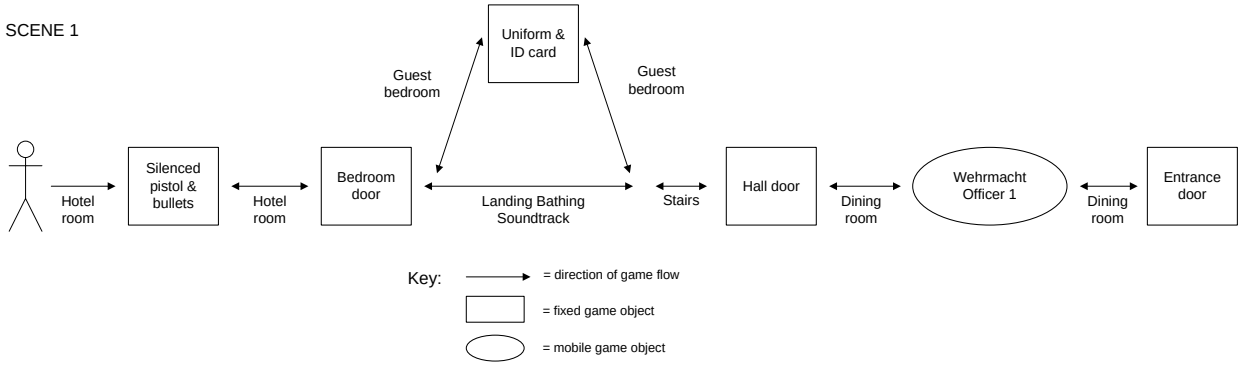

Figure 3.4: "On track" level of 'Medal of Honour: Frontline' represented with UML Use-case Diagram (Taylor, et al., 2006)

Natkin, Vega, & Grünvogel's (2004) approach to modelling game design addresses the need to model spatial-temporal relationships through the adaptation of *Petri-nets* and the use of generalised graphs called *hypergraphs*. They use special Petri-nets composed of three relations (*transaction a and transaction b are not related*, *transaction a is before transaction b*, and *if transaction a is executed then transaction b cannot be executed*) between two different transactions (actions of the player) to represent the temporal relationships (order of game sequence) in a game and game levels. Sets of transactions are then translated into a hypergraph to define the spatial relationship (events in relation to the map in the game world). In this approach, a circle is used to denote a place, a square or rectangle for transactions and arrows denote an input arc (which connects a place with a transaction) and output arc (connect transaction with place) (see Figure 3.5). Using this approach, game designers can model flow and events for game and game level. However, it cannot be used to document details about objects, characters and their behaviour, environment, user interface, narration or storytelling.

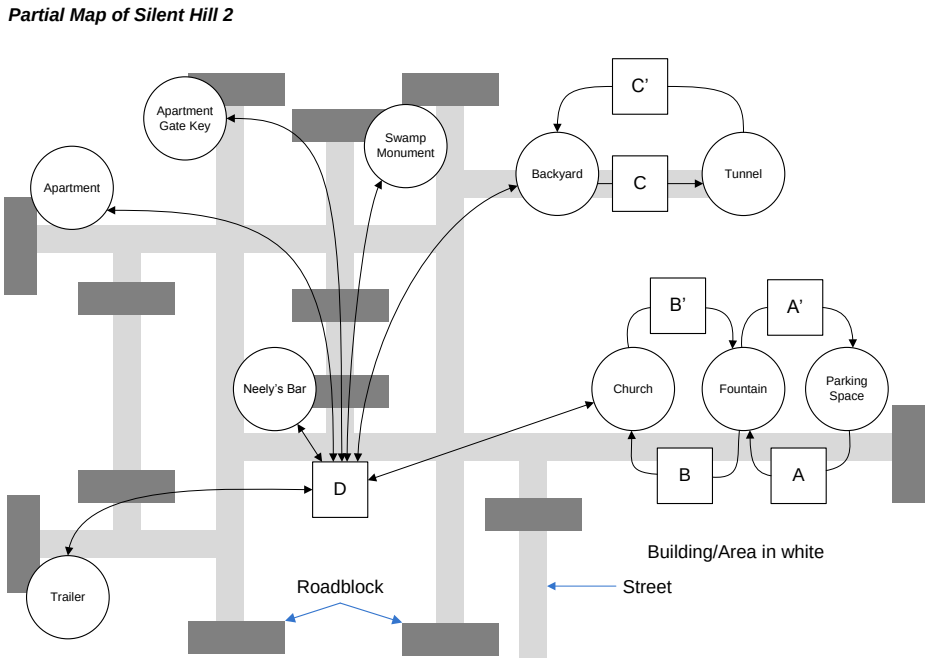

Figure 3.5: Partial Map of Silent Hill 2 represented in Hypergraph (Natkin, et al., 2004)

The formalised approaches of documenting game design reviewed only focus on documenting game flow formally using various graphical modelling languages, while other aspects of the game design remain informal and subjective for others to interpret using text description. Although these approaches help games designers in one or more ways, there are still many aspects of game design which can be documented formally which have yet to be studied while others such as visual, audio, narration, storytelling and user interface cannot be easily formalised and will continue to be documented using current approaches i.e. textual descriptions or sketches. It is highly desirable to formalise documentation of game design as it can enhance the effectiveness of translation of the design to software code.

3.2 Software Modelling Languages for Games

Modelling is a formalised process of documenting a design using syntactically composed reserved words or graphical notations. Similarly, the term modelling can be used for the approaches described in Section 3.1, but it refers to documenting aspects of game design. Through modelling, a software engineer captures the necessary requirements to produce a concrete model that best represents the software specifications of the game software. In addition, modelling allows the engineer to study problems and look for flaws in the design prior to development of the software system. Flowcharts, Data Flow Diagrams (DFD), Statecharts, Z-Notation, Petri-nets, Business Process Modelling Notation (BPMN), Integrated Computer-Aided Manufacturing Definition Language (IDEF) and UML are

some examples of software modelling languages that engineers can use to express the design of the game software.

Modelling was historically mathematically demanding and modelling languages were developed to suit programming languages during the early computing era such as FORTRAN, ALGOL, LISP and COBOL. As later generation programming languages such as C, Pascal and Ada which were designed to structure programs logically began to surface, software modelling took on a function-oriented paradigm. At present, Object-Oriented (OO) modelling is the de facto approach in the software industry as it promotes abstraction, encapsulation and reusability of software code.

Although most software modelling languages mentioned can be used to model aspects of game software, only a few are actually used in practice. *Tokenization* (Rollings & Morris, 2004) represents each game object (both interactive and non-interactive) as tokens. Interactions between tokens are represented in a triangular matrix as shown in Figure 3.6. Tokenization invites game developers to think in tokens and the associated interaction, but lacks many aspects necessary to represent the game software in its entirety. Simple casual games can be modelled using this approach without much complication. However, for modern computer games that offer more elaborate interactions and a greater number of tokens, this approach is not suitable as complexity of modelling increases. Another major setback for tokenization is evident during translation of model to software code. Although examples are available, there are information such as the game object's behaviour, setup of scenarios, flow of events and progressions that are still missing from the software model.

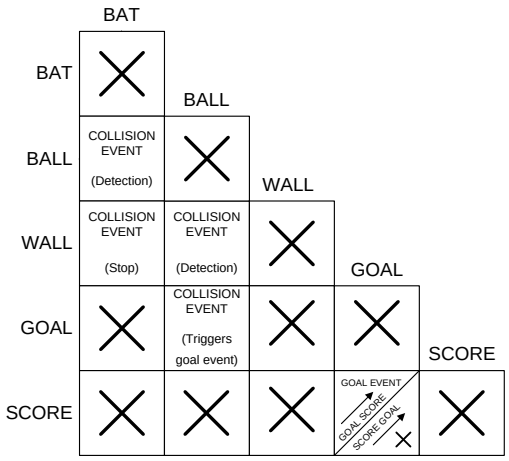

Figure 3.6: Tokenization Interaction Diagram of Pong Game (Rollings & Morris, 2004)

As many aspects of games deal with a collection of states that change based on interaction and rules, *Statecharts* or *Finite State Machine* (FSM) diagrams are ideal candidates for modelling games.

The *FSM* has been mostly used in modelling game AI (Kienzle, Denault, & Vangheluwe, 2007), although other game aspects such as character behaviour, game flow and game scenario can also be modelled with this approach. Rollings & Morris's (2004) approach is a pictorial version of FSM with slight modifications in notation for modelling a game object's behaviour (see Figure 3.7). A square is used to represent an individual state of a game object, while a circle denotes an event. Lines with a coloured dot at the end show the transition between states and the incoming event that triggers a transition. Coloured dots are used to pair the event to the transition between states in cases where there are more than one event referring to the same state. A circular arrow is introduced in the FSM diagram to include events that can cause entry to a state. Kienzle, Denault & Vangheluwe's (2007) approach uses a variant of *Rhapsody Statecharts* (Harel & Kugler, 2001) which combines statecharts and class diagrams to represent properties and behaviour of game objects (see Figure 3.8). In addition to class diagrams, statecharts are accompanied to switch or alter the state of a game object when certain conditions are met. Rounded rectangles represent the state while transition of state is shown using an arrow with conditions. Statecharts can be nested to create a multi-level statechart, but every statechart can only have a single begin state which is denoted by a filled black circle. Rhapsody Statecharts are a better approach than the pictorial FSM or classical statecharts as they represent both properties and methods of a software component and can easily use for representing other aspects of game software.

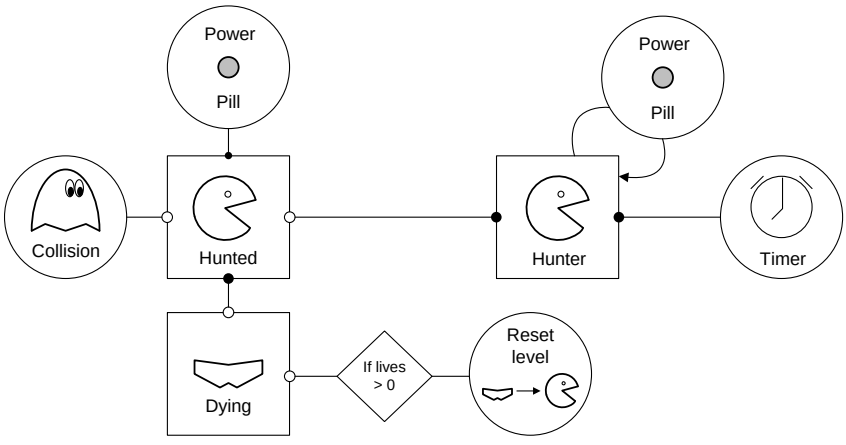

Figure 3.7: Statechart Diagram for Pacman game (Rollings & Morris, 2004)

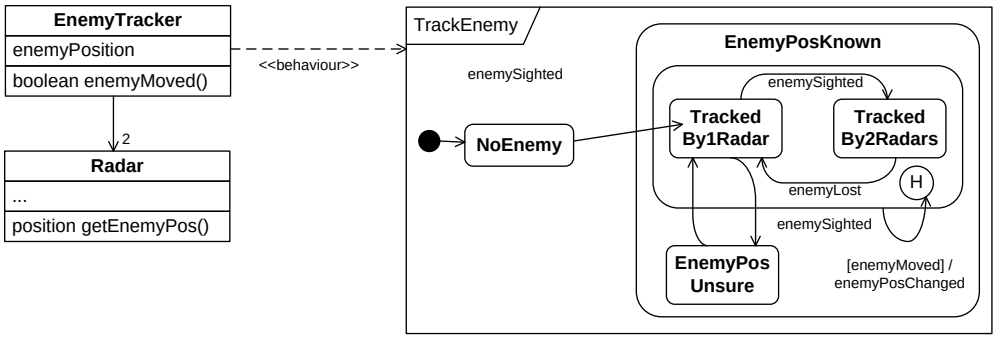

Figure 3.8: Rhapsody Statecharts of EnemyTracker AI component (Kienzle, et al., 2007)

As highlighted earlier, OO is the preferred approach and most computer games software is modelled using OO with the industry standard software diagramming language UML (Gal, Prado, Natkin, & Vega., 2002). This approach makes it simple for game developers to translate game software requirements to OO programming languages. UML consists of thirteen types of diagram to document three aspects of software systems: *functional*, *structural* and *behavioural*. There have been several proposals for modelling games software using UML. Ang & Rao (2004) represent player interaction with tokens using UML use case diagrams (see Figure 3.9) in an attempt to identify all possible interactions in the game before representing it in a class diagram. The identified interactions are behaviours of relevant classes in the game and it helps to discover other necessary classes to complete the software representation of the game. Bethke (2003) illustrates how a range of UML diagrams can effectively document design of a game software. For example, the UML use case diagram in Figure 3.10 is used to represent a player's interaction, while the UML class diagram featured in Figure 3.11 is used to represent game objects and static structure. Reyno & Cubel (2008) use UML class diagrams and state transition diagrams to represent structure and behaviour of game (see Figure 3.12). UML class diagrams can be used to represent the association between game objects represented as classes which can also be represented using *object diagrams* to examine the run-time static relationship behaviour. A collection of classes can be ordered into a high-level view using a component diagram, while a deployment diagram can be used for representing the physical arrangement of the game possibly utilising an external game engine. Activity diagrams can be an alternative to statecharts for modelling behaviour of game objects. In addition, sequence diagrams can be used to extend modelling of behaviour between game objects. UML certainly offers a range of useful diagramming approaches to document games software. The choice of UML diagram to use depends on how such diagramming tools help game developers to visualise the game software under construction and translate the specification to software code.

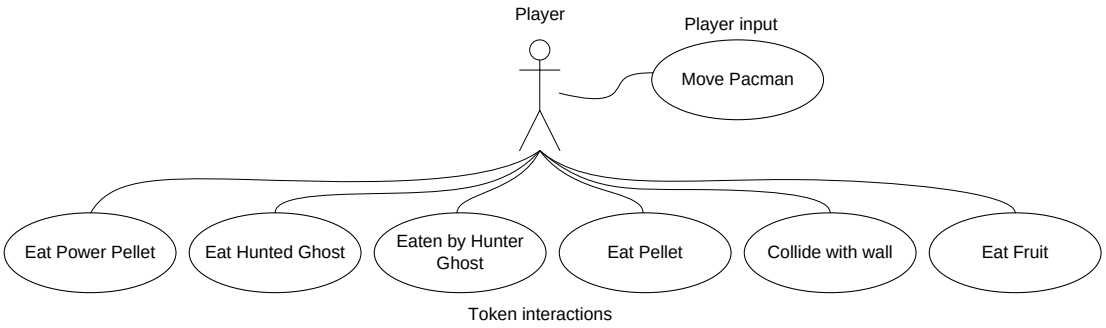

Figure 3.9: Use Case Diagram of Pacman Game (Ang & Rao, 2004)

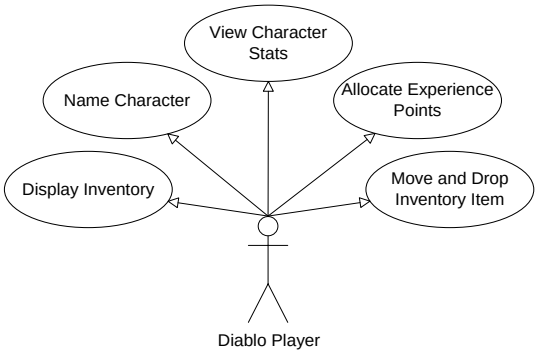

Figure 3.10: The character transaction use cases of Diablo (Bethke, 2003)

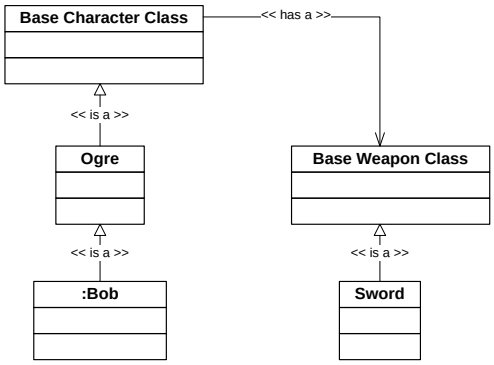

Figure 3.11: Example of Class Diagram (Bethke, 2003)

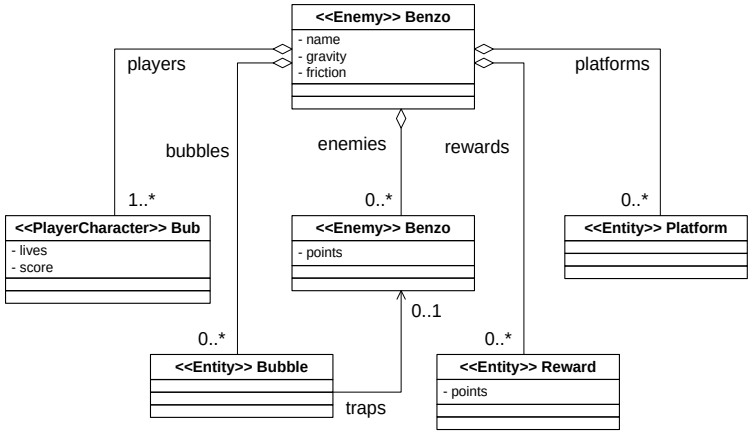

Figure 3.12: Structure Diagram of Bubble Bobble (Reyno & Cubel, 2008)

The reviewed game software modelling languages do not represent the entire collection of techniques for modelling game software. Many other techniques exist to model games software such as Flowcharts, BPMN, DFD and IDEF. In practice, a combination of software modelling techniques can be used to model the different aspects of game software as it is not always possible to represent all aspects of games software to be modelled using a single modelling language or notation.

3.3 Meta-Models for Games

Game meta-models are templates which game designers can use to complete the design of a game by providing the missing details required. They encase game technologies with purpose, rules, play and aesthetic representations (and pedagogy as well in the context of educational games) as creative and instructional expression of a knowledge domain. Game meta-models can be used with the aforementioned game design languages in Section 3.1 to guide amateurs and novices in producing game models that conform to the game meta-model. However, the completeness of a design specification is still very dependent on the attention to detail of the individual game designer. A game model provides the platform for game designers to complete the design of a game by simply providing the necessary details required by the game model.

In recent years, there are a growing number of formalised approaches developed to aid game designers to analyse, describe and study computer games. Hunicke, LeBlanc, & Zubek's (2004) Mechanics, Dynamics and Aesthetics (MDA) framework provides a guide for game designers to specify a game design systematically by focusing on a definition of aesthetic using a taxonomy for fun, the underlying system which supports the aesthetic requirements and finally the actions, behaviour and control mechanisms made available to game players. The taxonomy of fun consists of vocabularies such as *sensation*, *fantasy*, *narrative*, *challenge*, *fellowship*, *discovery*, *expression* and *submission* which, when used to describe play experiences, can help game designers think about the underlying dynamics and mechanics of a game. It is an interesting approach to design games in a structured manner while still fostering creativity. However, it only focuses on the aspects relating to game-play and has no solid structure to formally represent the components that make up a game such as game objects, characters, environment, user interfaces, storytelling, narration and game progression.

Björk, Lundgren & Holopainen (2003) approach game design from a research-driven perspective by cataloguing descriptions of recurring interactions that relate to game-play which they term as *game design patterns* for analysing and studying games. Each game design pattern has a name, core definition of the pattern, a general description, usage description of the pattern, a description on the consequences of this game design pattern, a description of the relationships between other game design patterns and references that describe the origin of the pattern. To date there are more than 200 patterns identified and these are organised into collections of *game elements; resource and resource management; information, management and presentation; action and events; narrative structures,*

predictability and immersion; social interaction; goals; goal structure; game session; game mastery and balancing; and *meta game, replayability and learning curves* (Björk & Holopainen, 2004). The use of patterns and templates to document aspects of game design is a formalised attempt and it promotes better organisation and structure in terms of documenting details on game play compared to MDA for example. Although the original intention of this approach is for analysing and studying existing games, the collection of game design patterns can now be used as a reference and model for defining components of future games.

The Game Ontology Project (GOP) (Zagal, Mateas, Fernández-Vara, Hochhalter, & Lichti, 2005) addresses the need for formalised game design through the identification of both common and distinct elements of game design abstractly, and orders these hierarchically to form a game ontology. The game ontology is regarded as a language for game design and also a framework for game designers to improve their understanding on what to design in computer games. At the highest-level, the ontology focuses on the definition of *interfaces* that map user inputs to a set of allowable actions of an avatar; *rules* that govern the interactions and economies in the game world; *goals* that determine a player's success or failure in the game; *entities* that populate the game world; and *entity manipulations* which define how entities behave physically in the game world. Each ontology entry is labelled with a name, listed with relationships with other elements in the hierarchy and given a description about the element as well as examples of both concrete and partial reification of the elements. Game designers can make references to the ontology and design games by filling the descriptions of the game design. Formalising the descriptions of the ontology entries are still necessary to ensure such a game model can be translated to software code.

Rapid Analysis Method (RAM) (Järvinen, 2007) is a set of methods for analysis and design of computer games from a system viewpoint and players' viewpoint focusing on the emotional and the socio-physiological aspects of game-play. It consists of seven tools to aid amateur game designers to systematically identify and analyse game elements, game mechanics and the goal they are related to, permissible interactions, eliciting conditions for emotions during game-play, game rhetoric and to study the anatomy of a game. RAM is a comprehensive approach for studying games and is suitable for use as a game meta-model. Nevertheless, there are still missing details required for describing computer games in terms of attributes of game elements instead of building a taxonomy for cataloguing common game elements.

Fullerton's (2008) model represents a computer game from a system context by studying the formalised elements and dramatic elements that compose a game. *Objectives*, *procedures*, *rules*, *resources*, *conflict*, *boundary* and *outcome* are the formalised elements that define the structure of a game, whereas *play*, *challenge*, *premise*, *character* and *story* are the dramatic elements used to encase the formalised elements in order to engage the game players. This is a rather comprehensive representation of computer games that encompasses both technical and artistic aspects of game design, but yet is still incomplete. There is one dramatic element missing: *game objects* that fill the game world. It may have most of the elements to represent a game but it lacks the formalised language necessary to describe these elements accurately.

Nelson & Mateas' (2007) model focuses on four domains: *abstract game mechanics* which regulate the operational aspects of a computer game, *concrete game representation* which presents the abstract game mechanics through audio-visual means, *thematic content* which are graphics and sounds created to represent the game world, and *control mapping* which relates the player's input to methods that modify abstract game state. The simplicity of the model helps one to understand the computer game as a system easily, but it is a rather brief model to represent complete games as many aspects of game design remain widely undefined making it a less favourable candidate for a game meta-model.

The Narrative, Entertainment, Simulation and Interaction (NESI) model proposed by Sarinho & Apolinário (2008) is a game feature model. It organises four main features of game design into a hierarchical order: *narrative* which consists of flow that progresses the game with different goals governed by rules that determines the success or failure of player; *entertainment* which is made up of a theme for the game and the player's immersive experience during the game play; *simulation* which is composed of game elements such as characters, items and buildings that populate the game world and the relationships among them; and finally, *interaction* that maps the game player's control to the game-play features and the presentation of the game through audio, visual or haptic devices. The NESI model is another possible candidate for a game meta-model.

The game meta-models reviewed provide interesting ways to document and analyse game design. Each of these has a different focus on aspects of game design and approach to documenting the details of game design. Among all the approaches reviewed, the MDA is probably the least qualified candidate as a game meta-model due to the lack of structure and identification of elements that represent games in their entirety. Nevertheless, it is a useful tool that can aid the game designer to think creatively when designing fun and engaging game-play. Contrary to that, the game design

patterns approach is a good candidate for a game meta-model. Game design patterns provide a template for documenting the multiple aspects of game design and organises these into collections which can be used as references for novices and amateur game designers to create their own games. It, however, lacks formalised descriptions of game design patterns required in MDE. The GOP has a more logical organisation of the elements of games compared to game design patterns and demands relational information which is crucial for writing software code. Similarly, it lacks formalisation in describing each ontology entry. The RAM has a good method to identify, analyse and study game elements, but it lacks a method to formally describe these elements. Nelson & Mateas' (2007) model promotes simplicity in describing a game but lacks refined structure, categories in design details and a formalised method of describing the game. The NESI model is a useful approach to describe games through description of features, but it falls short on how one can formally describe these features. By 'formalised' we mean the use of a single unified language that may be human or machine-readable. Fullerton's (2008) model is by far the most comprehensive model reviewed, but it still lacks one element (i.e. game objects) and uses creative writing to document these elements. It is clear that the one common shortcoming these approaches share is the lack of formality and details to accurately represent a computer game. Such an issue must be addressed to ensure a game model produced using a game meta-model can be mapped to a game software model during the process of model transformation in a model driven game development framework.

3.4 Software Models for Games

A game software model, as opposed to a game meta-model, represents computer games from a computational perspective independent of any technology platform. From an MDE perspective, it is translated from the game model to specify the data, methods, processes and technologies that comprises the game software. Depending on the game genre, a computer game may require different types of visual renderer, AI, game physics and other software components to realise the game design. These technologies are usually packaged into a game software framework which is used to produce a variety of different computer games. It is rather challenging to have one game software framework to represent all computer games across different genres due to technical constraints that game software frameworks inherit. Although there are differences in technologies used, the basic structure of computer games software remains similar across different genres. It is, therefore, crucial that a game software model has a high degree of abstraction to allow it to cater for a range of game genres.

A game software model is dependent on the choice of programming paradigm. Early computer games were programmed with languages such as BASIC and C which support both unstructured and structured programming paradigms. Today most game developers would opt to engineer games software using an OO model as it offers versatility and reusability of code which can increase productivity and ease maintenance of code. Most OO-influenced models represent game software based on its processes or game components. The *SharpLudus Ontology* (Furtado & Santos, 2006a) is a game software model built onto six key concepts that represent computer games, namely *configuration*, *graphics*, *entities*, *event*, *flow* and *audio*. Each of these concepts is elaborated in detail to represent the properties and methods that construct a 2D adventure game. Although SharpLudus Ontology is designed specifically for the 2D adventure game genre, the software model can also be used for other 2D games that are built using tile-based worlds.

The *GameSystem, DecisionSupport and SceneView* (GDS) model (Sarinho & Apolinário, 2009) represents game software with a combination of software features organised based on its functionality in computer game. Each feature describes generic configurations and behavioural aspects for a specific part of the game software. Features in GameSystem monitor both game events and inputs from the game-player, and trigger the associated actions that can affect all defined data including those defined in DecisionSupport and SceneView. DecisionSupport groups a selection of game AI features that provides the intelligence for non-player characters (NPC), while SceneView describes the features that observe events within a defined spatial environment and those affecting the presentation of the game such as audio, graphics and physics. This shares many characteristics of the MVC (Model-View-Controller) design pattern (Veit & Herrmann, 2003).

Altunbay, Cetinkaya, & Metin's (2009) *board game model* is a representation of game software specific to board games. The *board game model* is based on the concepts of: *Game Engine*, *Game Element*, *Player*, *Event*, *Action*, *Game State*, *Goal*, *Sub-Goal*, *Non-Movable Element*, *Movable Element* and *Rules*. Most of these concepts are very specific towards the board game genre except for *Player*, *Goal* and *Rules* which are generic for all games genres.

The SharpLudus Ontology, board game model and GDS each have their own merits and focus in representing games software. The key concepts introduced in SharpLudus Ontology are a useful guide for representing other forms of game software. It represents the essential components in game software but it will need to be extended to support the requirements for other game software before it is deemed useful. The board game model is comprehensive but rather specific to a single game genre

thereby limiting its usefulness. The GDS model is a credible candidate as it generically and comprehensively represents game software based on features which can be mapped across to a game software framework and, therefore, can represent a broader range of computer games software. Moreover, it clearly separates games software into MVC components (GameSystem as the model, SceneView as the view and DecisionSupport as the controller) model permitting independent development, testing and maintenance of each component. It is apparent that designing a game software model requires good understanding of game processes and the game software framework. As current game development trends set such a high-dependency on game software frameworks to increase productivity, it is important that a game software model structures game design requirements logically and maps to common functionalities of a game software framework.

3.5 Game Software Frameworks

A game software framework (also known in the game industry as a game engine) is a set of Application Programming Interfaces (APIs). It consists of software components that perform graphic rendering (2D or 3D), game physics computation such as collision detection, collision reaction and locomotion, programmed intelligence, user input, game data management and other supporting technologies to operate the game software. These software components are built to manage, accept, compute and communicate data with other software components. An appropriate API can increase the productivity of game developers allowing them to reuse proven and tested software code to produce a variety of computer games within the scope of the software framework. Game software frameworks vary depending on technology features that are often constrained by particular game genres, technology platforms (hardware) and visual dimension of the game world (2D or 3D), but technology components such as graphics, sound, user input, basic game physics and user interfaces remain essential across all game software frameworks. For instance, the *Unreal engine*[9] is suitable for development of action or adventure games in first or third person view but it will require modification to include necessary physics code if it is to be used to support sports game genres. Not all game software frameworks support all features since integrating these technological components under a single framework is a highly complex and expensive task. Furthermore, not all computer games software will require the entire collection of software components to operate. Budget, support, features offered, ease of use, suitability, royalties and targeted hardware platform are some examples of

[9] Read more about Unreal Engine from http://www.unrealtechnology.com/.

key criteria which can help to decide the most suitable game software framework for a project. Identifying features supported and techniques used in game software frameworks can help to provide valuable information for MDE developers to design a framework that is robust and interoperable.

There are many game software frameworks developed to date (at the time of writing, there are 297 3D game engines registered at the DevMaster.net[10] website). In general, these game software frameworks can be categorised from a purely financial point of view as being one of *commercial, open source* and *proprietary*, or using the visual dimension they cater for i.e. 2D or 3D.

Commercial game software frameworks are much more stable and offer a reliable (supported) option over open source game software frameworks. The *Torque Game Engine* (TGE) is an example of royalty-free game engine which is offered to developers at a very affordable price compared to other commercial game software frameworks, in which TGE developers only pay for the technology once and are allowed to develop as many games software applications as they wish. *Unreal engine*, *Source engine*, *CryEngine* and *Id Tech* (formerly known as *Quake Engine*) are some notable examples of highly sophisticated game technologies that can help developers to produce high-quality games. These premium game technologies are available for licensing and require a one-off royalty payment (per-title fee) or royalty for every copy of the game sold, or both. However, some of these are made available for free for educational purposes such as the *3DSTATE 3D Engine*, *XNA*, *CryEngine 3*, *Unreal Development Kit* (UDK) and *Unity 3D*.

Open source game software frameworks are developed for very specific purposes and often made available to the public when they reach the end of their useful lifecycle. Examples of well-known open source game software frameworks include *Apocalyx Engine, Baja Engine, Blender Game Engine, Irrlicht Engine, jME, jPCT, Lilith3D, Object Oriented Graphic Rendering Engine* (OGRE), *OpenSceneGraph, Panda3D* and *The Nebula Device 3*. Most of these game software frameworks are distributed under an open source license which allows game developers to produce games software for both commercial and non-commercial purposes at no cost. Some open source game software frameworks only implement a limited range of techniques and some fail to address certain features, for example *Panda 3D* does not provide solution shadows, scene management and curves. Game developers can still achieve the shadows effect using *Panda 3D* either by adding this feature to the framework themselves (this, however, requires a specialist skill) or by creatively using textures and shaders to achieve the desired shadow effects. However, there is little or sometimes no support at all

[10] See known 3D game engine listing at http://www.devmaster.net/engines/list.php.

for customising some technologies to fit the use of a project with a different specification. For more details of these game software frameworks mentioned refer to Table 3.1.

Table 3.1: Details on Commercial and Open source Game Software Framework

Game Engine	License	URL
3DSTATE 3D	Open source	http://www.3dstate.com/
Unreal Engine	Commercial & the Unreal Development Kits (UDK) available free for educational purpose	http://www.unrealtechnology.com/ http://www.udk.com/
Source Engine	Commercial	http://source.valvesoftware.com/index.php.
CryEngine	Commercial & available free for educational purpose.	http://www.crytek.com/
Id Tech (formally known as Quake Engine)	Commercial	http://www.idsoftware.com/business/idtech3/
XNA	Commercial & available free for educational purpose	http://www.xna.com/
Unity 3D	Commercial & available free for educational purpose	http://unity3d.com/unity
Apocalyx Engine	Open source	http://apocalyx.sourceforge.net/
Baja Engine	Open source	http://www.bajaengine.com/
Irrlicht Engine	Open source	http://irrlicht.sourceforge.net/
Java Monkey Engine (jME)	Open source	http://www.jmonkeyengine.com/
jPCT	Open source	http://www.jpct.net/
Lilith3D	Open source	http://www.grinninglizard.com/lilith/
Oriented Graphic Rendering Engine (ORGE),	Open source	http://www.ogre3d.org/
OpenSceneGraph	Open source	http://www.openscenegraph.org/projects/osg
Panda3D	Open source	http://www.panda3d.org/
The Nebula Device 3	Open source	http://nebuladevice.cubik.org/

Proprietary game software frameworks are created in-house (by game development companies) solely for internal usage. These are usually on a par with commercial game software frameworks and often of more superior quality. *RenderWare*, technology produced by Criterion Games and acquired by Electronic Arts (EA) is used in many games published by EA such as the Need For Speed and Burnout series; *EGO* developed in-house at Codemasters in the UK is used in the production of a range of game titles including Colin McRae: Dirt, Race Driver: Grid and Operation Flashpoint: Dragon Rising; *Jade Engine* Ubisoft's in-house game technology is used in games such as Rayman Raving Rabbids I and II, and Teenage Mutant Ninja Turtles. These technologies are mainly used for in-house game titles.

Based on the ratings gathered from the devmaster.net forum and our analysis on features offered by each game software framework, we rank these game software frameworks based on popularity and sophistication of features as shown in Figure 3.13 to give a general idea of the current technology

landscape. Each game software framework is evaluated in terms of *sophistication of features*, *support for developers* and *quality impact* and *feature-richness*, and rank based on scores obtained for each aspects evaluated. The features and techniques used in each game software framework are identified from the devmaster.net forum and scores are awarded on the basis of availability and the importance of a feature or technique. The score for sophistication of feature is evaluated based on *uniqueness of the technique* (out of 5 for each technique supported), *extensibility of technique* (out of 5 for each technique supported) and *game-significance of the feature* (out of 5). Aspect of support is scored based on *sophistication of toolset* (out of 20), *knowledge-based support* (out of 20) and *cross-platform deployment* (out of 10), while *quality impact* is scored out of 20. These scores are given based on ratings given on devmasters.com. It is important to note that this survey is done independently to show the criteria that affect the selection of game software framework in general. The raw results of this survey are tabulated in Table 3.2 and Table 3.3. It is apparent that popular game software frameworks such as Panda3D, XNA, Unity 3D, CryEngine 3 do not necessarily offer more features compared to the less popular game software frameworks like the 3DSTATE 3D Engine and the Nebula Device 3. Some of the reasons these less-sophisticated game software frameworks gain popularity amongst the developers includes the range of technical support, availability of high-level toolsets, large community base for exchange of knowledge and stability of the game software framework.

Table 3.2: Evaluation of Features for Open Source and Royalty-free 3D Game Software Frameworks

Game Software Framework	Features															
	Lighting	Shadows	Texturing	Shaders	Rendering	Scene Management	Animation	Meshes	Surfaces & Curves	Special Effects	Terrain	Networking System	Sound	Physics	Artificial Intelligence	SCORES
3DSTATE 3D Engine	25	13	29	19	22	4	15	8	-	17	6	6	12	7	-	183
Apocalyx Engine	6	5	11	10	-	6	8	-	8	22	6	3	7	7	-	99
Baja Engine	22	-	13	16	11	-	6	6	-	31	6	9	12	7	5	144
Blender Game Engine	16	5	15	16	11	4	11	8	8	13	6	5	8	15	5	146
Crystal Space	12	12	17	16	10	10	8	9	13	15	9	-	8	10	-	137
CryEngine 3	21	3	15	15	10	4	8	6	-	17	4	3	4	11	9	130
Cube 2: Sauerbraten	14	5	17	16	11	10	10	6	-	16	6	8	12	7	17	155
Delta3D	16	-	21	19	25	9	6	11	-	17	9	6	8	9	-	156
G3D	10	8	14	16	10	4	6	6	-	7	6	-	-	6	-	93
Irrlicht Engine	18	7	20	19	20	10	13	6	-	5	9	-	-	6	-	133
jME	11	7	22	19	20	8	8	8	8	24	9	-	7	7	-	158
jPCT	5	5	16	-	15	11	8	6	-	14	-	-	-	5	-	85
Lilith3D	10	7	10	16	6	6	6	11	-	17	6	-	-	-	-	95
OGRE	17	8	30	19	20	13	13	11	8	21	-	-	-	10	-	170
OpenSceneGraph	14	12	20	19	15	7	10	8	-	12	9	-	13	10	-	149
Panda3D	10	-	6	7	14	-	6	8	-	8	6	5	13	15	8	106
The Nebula Device 2	21	-	25	19	12	13	14	8	13	13	14	6	13	10	5	186
TGE	12	12	17	-	20	10	13	16	-	21	9	5	13	15	10	173
UDK	30	12	35	19	15	12	20	21	13	41	14	6	13	15	18	284
Unity 3D	-	9	14	19	16	-	6	8	-	14	6	5	13	15	5	130

Table 3.3: Evaluation of Open Source and Royalty-free 3D Game Software Frameworks

Game Software Framework	Criteria						
	Sophistication of Features	Sophistication of Toolset	Knowledge-based Support	Multiplatform Support	Quality Impact	TOTAL SCORES	RANK
3DSTATE 3D Engine	183	-	12	6	8	**209**	**5**
Apocalyx Engine	99	6	16	3	7	**131**	**16**
Baja Engine	144	-	14	6	7	**171**	**15**
Blender Game Engine	146	16	16	10	7	**195**	**8**
Crystal Space	137	-	16	10	14	**177**	**14**
CryEngine 3	130	10	18	13	20	**191**	**9**
Cube 2: Sauerbraten	155	-	11	10	14	**190**	**10**
Delta3D	156	-	16	6	10	**188**	**12**
G3D	93	-	16	10	12	**131**	**16**
Irrlicht Engine	133	7	16	13	11	**180**	**13**
jME	158	4	18	10	15	**205**	**6**
jPCT	85	-	18	10	8	**121**	**18**
Lilith3D	95	-	11	6	6	**118**	**19**
OGRE	170	-	16	10	18	**214**	**3**
OpenSceneGraph	149	3	14	12	12	**190**	**10**
Panda3D	106	6	17	10	12	**151**	**15**
The Nebula Device 2	186	-	13	3	12	**214**	**3**
TGE	173	16	16	20	20	**245**	**2**
UDK	284	20	17	16	20	**357**	**1**
Unity 3D	130	18	18	18	20	**204**	**7**

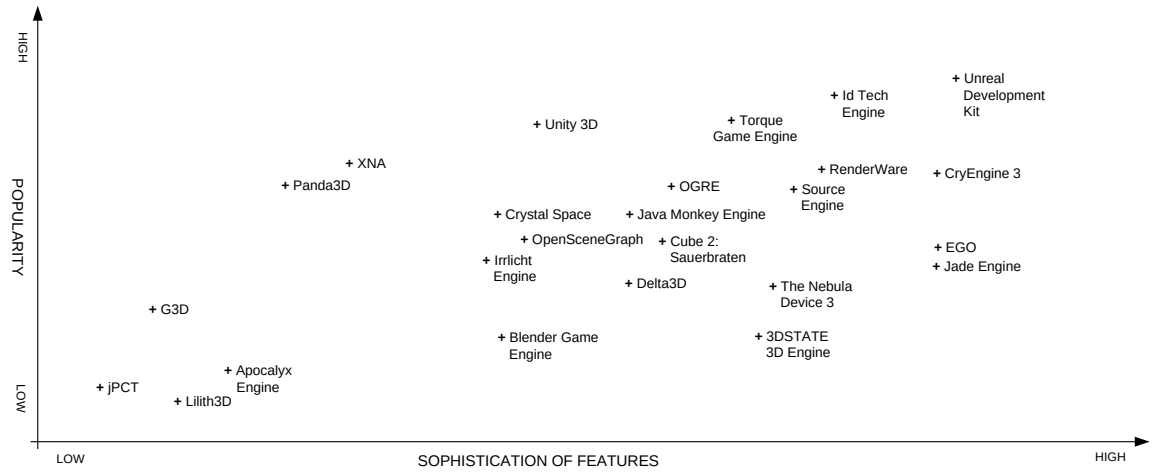

Figure 3.13: Game Software Frameworks ranked according to Popularity and Sophistication of Features

One of the most crucial components for game software is the graphics which provide the visualisation of the game world to game players. Clever usage of graphic technologies can captivate a game player's interest and provide an immersive game play experience. In general, aspects related to graphic technologies can be categorised into lighting, shadows, texturing, shaders, scene management, meshes, terrain, special effects (particle systems) and rendering. Each of these aforementioned categories has its own collection of specialised techniques to achieve the desired visual results without compromising the frame rate (see Table 3.2). In fact most of the game software frameworks are essentially real-time 3D graphics engines that have been extended to form a game software framework. For example, game physics engines such as the Newton Game Dynamics[11], Open Dynamic Engine[12] or Havok Physics[13], game AI engines such as Havok AI[14] and network components for games such as Game Networking Engine[15] (GNE) or RakNet[16] can be added to extend any 3D graphics engine. The remaining features which are not available can be developed in-house by specialist game software engineers to complete the game software framework. Physics, AI, animation, networking and sound have a variety of techniques available to support game development (see Table 3.3). In addition, some game software frameworks provide a collection of tools such as a level-editor, visual sound tools and animation tools to allow non-technical members of the team to fine-tune games visually instead of directly editing the source code. This review of game software framework

[11] Read more about Newton Game Dynamics from http://newtondynamics.com/forum/newton.php.
[12] Read more about Open Dynamic Engine from http://www.ode.org/.
[13] Read more about Havok Physics from http://www.havok.com/index.php?page=havok-physics.
[14] Read more about Havok AI from http://www.havok.com/index.php?page=havok-ai.
[15] Read more about GNE from http://www.gillius.org/gne/.
[16] Read more about RakNet from http://www.jenkinssoftware.com/raknet/index.html.

features and their associated techniques provides an overview of the level of support a game software model should offer to support a range of game platforms. Therefore, it is crucial that any game software model can be extended easily over time to remain heterogeneous among the game software frameworks. To achieve this design requirement, it is necessary that the game software model be highly abstracted else it can only support a limited number of game software frameworks thereby constraining the variety of computer games which can be built with future game software frameworks.

Table 3.4: List of Techniques for Graphic Features.

Graphic Features	Techniques
Lighting	Light mapping, gloss mapping, per-pixel lighting, per-vertex lighting, anisotropic filtering and radiosity.
Shadows	Shadowmap, projected planar and shadow volume.
Texturing	Basic, multi-texturing, bump mapping, mip mapping, volumetric texturing, projective texture mapping and procedural texturing.
Shaders	Vertex shader, pixel shader and high-level shader.
Scene Management	General, Binary Space Partition (BSP), Potentially Visible Set (PVS), portals, occlusion culling, octrees and Level of Details (LOD).
Meshes	Mesh loading, mesh skinning, progressive mesh, tessellation, deformation
Terrain	Rendering, Continuous Level of Details (CLOD) and splatting.
Special Effects	Environment mapping, billboard, sky, water, fog, mirror, lens flares, particle systems, fire, explosion, decals, weather, motion blur and depth of field.
Surfaces & Curves	Spline and Patches.
Rendering	Fixed-function, stereo rendering, render-to-texture, fonts and Graphical User Interface (GUI).

Table 3.5: List of Techniques for Other Game Feature.

Other Features	Techniques
Physics	Basic physics, collision detection, rigid body and vehicle physics.
AI	Scripted, path-finding, finite state machines, decision making, neural network, fuzzy logic and etc.
Animation	Keyframe Animation, inverse kinematics, forward kinematics, skeletal animation, facial animation, animation blending and morphing.
Network	Peer-to-peer, master-server, client-server, communication (text, voice and video)
Sound	2D sound, 3D sound, streaming sound, mixing, digital signal processing effects and filtering.

3.6 Model-Driven Development Frameworks for Games

Supporting any MDE practice is a framework that unifies models and manages the transformation between these models using appropriate MDE tools to produce the desired software artefact. The OMG's Model Driven Architecture (MDA)(OMG, 2001), the Domain-Driven Software Development Framework (Agrawal, Karsai, & Ledeczi, 2003) and Modelling Turnpike (mTurnpike) (Wada & Suzuki, 2006) are examples of model-driven frameworks that aid software architects to develop their own MDE solution to suit a particular domain. These model-driven frameworks have been adapted to suit domains such as security (Basin, Doser, & Lodderstedt, 2006), content repurposing (Obrenovic,

et al., 2004), software testing (Javed, Strooper, & Watson, 2007) and pervasive computing (Hemme-Unger, Flor, & Vogler, 2003).

In the game development domain, the use of software frameworks and tools are usual practice amongst commercial game developers. Although current practice does improve productivity of the development team while providing maximum control and flexibility to artistically craft the game software, the production pipeline is still very reliant on specialist artists and programmers. This is rarely an issue in the commercial sector where budgets are supported by a business case. The use of MDE in games development is still in its infancy. The growing interest amongst the game enthusiast and practioners of serious game that wants to make their own games, MDE can provide the technological solution to aid them to produce games with minimal reliance on professional game developers.

The *SharpLudus Game Software Factory* (Furtado & Santos, 2006b) is an early attempt of a model-driven approach to increase productivity of game development teams in developing 2D adventure games. The framework consists of (Furtado, 2006);

- A domain-specific modelling language - SharpLudus Game Modelling Language (SLGML) that allows the game designer to model the flow of the game using a room designer and info display designer;
- A semantic validator that checks the model to ensure the design conforms to the semantics of SLGML, and finally;
- A code generator built on top of Microsoft's Visual Studio Integrated Development Environment (IDE) targeted towards generation of C# code for the XNA game engine.

Reyno & Cubel's (2008) *Model-Driven Game Development* (MDGD) approach introduces the use of a selection of UML diagrams to gather required information to automate generation of code for 2D platform games. The framework comprises:

- Two Platform Independent Models (PIM); A UML Class Diagram extended with stereotypes is used to model the relationship between different game entities[17] within the game world while a UML State Transition Diagram is use to model behaviour of the game entity;
- A Platform Specific Model (PSM) to map game actions to hardware controls, and;

[17] Game entity refers to objects within the game world. This term is synonymous with game object in this paper.

- A transformation tool to translate the models into C++ source code compliant to the Haaf Game Engine[18].

Altunbay, Cetinkaya, & Metin's (2009) model-driven approach for developing board games shares some similarity with the MDGD approach. It uses the UML class diagram as the modelling language to represent the game model. The framework is comprised of:

- A Board Game Meta-model represented using a UML Class Diagram;
- A Game Domain Specific Language[19] (GameDSL) - a meta-model developed specifically to aid the definition of game logic;
- A model-to-model transformation tool to transform the Board Game Model onto GameDSL and model-to-text transformation tool which subsequently transforms the Board Game Model in GameDSL format to Java source code.

The SharpLudus Game Software Factory, MDGD and Board Game Framework are research-driven attempts to adopt a model-driven approach towards developing 2D games. These frameworks may seem to share similar architectures but they differ in terms of models, tools and targeted platform. These frameworks are also developed for a specific game genre and therefore adapting these frameworks to suit other genres would require redefinition of models and new generators to facilitate transformation of model to code. It indicates that a model-driven game framework that can support multiple game software frameworks and is easily extensible is highly desirable to drive the production of a range of computer games which can be played on a range platform. Architecting such a framework to bind both the game model and game software model is a challenge and will require extensive research on game design, game development and a strong understanding of game software frameworks.

3.7 Model-Driven Development Environment for Games

Model-driven game development environments are a realisation of model-driven game development frameworks and associated toolset that allow game developers to take advantage of these technologies to accelerate game development. These tools may resemble some of the tools used by the industry (for example, a game editor or level editor) both in functionality and interface but are presented as simplified interfaces for the game designer to describe the game models in a structured and orderly manner.

[18] Read more about Haaf Game Engine from http://hge.relishgames.com/.
[19] Read more about GameDSL from http://gamedsl.tuxfamily.org/.

As described in Section 3.6, the SharpLudus Game Software Factory offers a range of tools to assist game designers in modelling different aspects of 2D adventure games. The development environment is a customised version of the Microsoft Visual Studio IDE that allows game designers to systematically define the game through a collection of wizard-based user interfaces. The *SLMGL Visual Editor* (see Figure 3.14) provides a visual drag-and-drop environment for modelling game flow and a range of form-based user interfaces to create the representation of game entities, design the screen, design the layout of the room (a game level), group collections of images as sprites, link audio files, and define conditions for game states, events and event feedback in the game (see Figure 3.15). These interfaces are viewports into the game model which allows game designers to effectively concentrate on a specific aspect of the game to avoid information overload. To ensure game models conform to the semantics of SLGML, a semantic validator is incorporated into the development environment to alert designers about errors in modelling before code is generated. The SharpLudus Game Software Factory uses the *Microsoft Domain Specific Language (DSL) Tools*[20] to implement its code generator by scripting the transformation rules to map the model to generate the code compliant to the game software framework used in the project (Furtado, 2006).

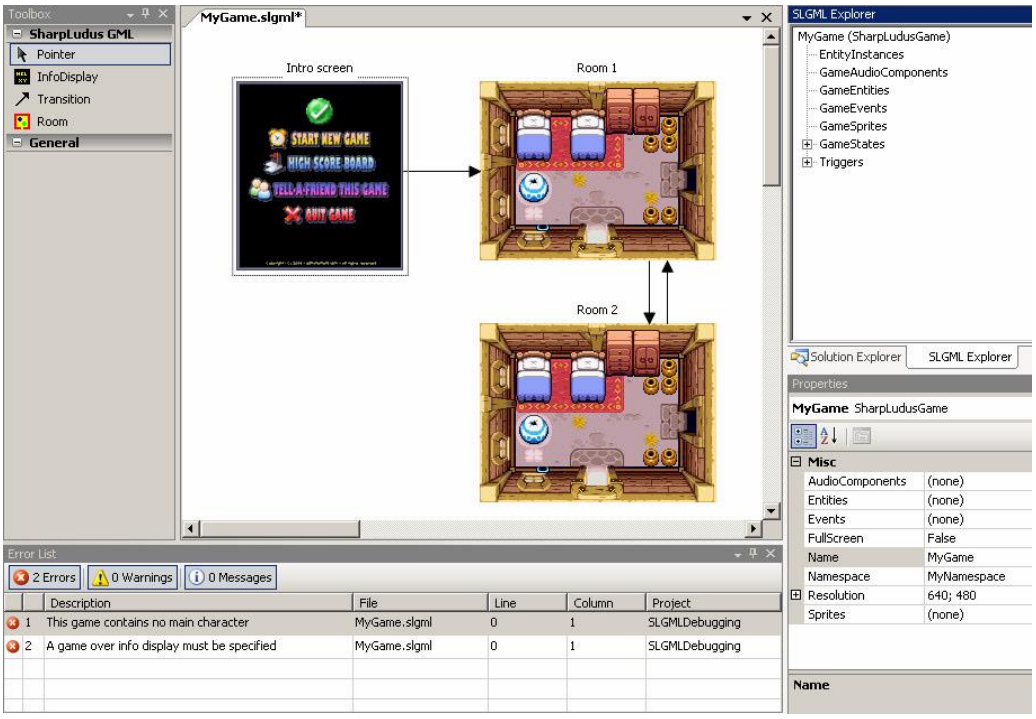

Figure 3.14: SLGML Modelling Experience (Furtado, 2006)

[20] Microsoft DSL Tools is a visualisation and modelling software development kit made available only for the Microsoft Visual Studio IDE to enable software engineers to build custom visual modelling environment to be hosted within the Microsoft Visual Studio IDE. Read more about Microsoft DSL Tools from http://code.msdn.microsoft.com/DSLToolsLab.

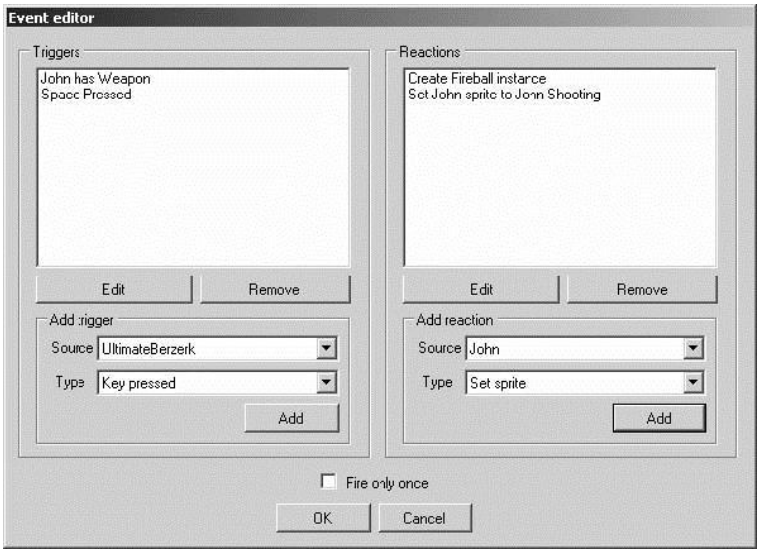

Figure 3.15: Event Editor Wizard in SharpLudus Game Factory (Furtado, 2006)

The game development environment used by Reyno & Cubel (2008) is implemented using the *Eclipse Modelling Framework*[21] (EMF) which allows the user to model 2D platform games using UML diagrams. The modelling tool provides the standard UML notations to model structure in the form of a class diagram, behaviour of a game entity using a state diagram and control mapping using an activity-like diagram. Transformation from these diagrams to code is implemented using *MOFScript*[22] to generate C++ code compatible with the *Haaf Game Engine*. *MOFScript* is a tool for text transformation that generates text or code from any MOF-based model such as UML.

Altunbay, Cetinkaya, & Metin (2009) use a similar approach as Reyno & Cubel in using a modelling environment built on EMF. A board game model is then translated to GameDSL using the *Atlas Transformation Language*[23] (ATL) model transformation tool and finally transformed into Java source code using *MOFScript*.

Amongst the reviewed model-driven game development environments, the SLGML modelling environment is a promising example of a game modelling environment. This is mainly due to the choice of DSML and interfaces used. The use of domain specific notations instead of the industry standard UML notations is a better choice as it makes games modelling more relevant to the users. Although UML provides more flexibility in modelling games software, it is rather technical and unguided. Only experienced users who have a good understanding of game software would be able to model a game correctly. It is also important to balance the use of visual notations and other forms of

[21] Read more about EMF from http://www.eclipse.org/modeling/emf/.
[22] Read more about MOFScript from http://www.eclipse.org/gmt/mofscript/.
[23] Read more about ATL from http://www.eclipse.org/atl/.

user interface for users to model a game. An appropriate interface can increase productivity and reduce the learning curve of using the development environment. Therefore it is crucial that model-driven educational games development environments use DSML notations suited for the domain and a collection of interfaces that encapsulate the technical aspects of game development from domain experts.

3.8 Model-Driven Engineering Technologies

Research in MDE is advancing rapidly and now there is a range of integrated environments and code frameworks available to support MDE practitioners in adoption of model-driven development approach. Integrated environments can provide a complete solution where the practitioner can define a DSL and its transformation rules to generate the desired software artefacts and host the modelling component within the same environment, while a code framework provides the basis for development of MDE tools to support the MDD.

3.8.1 Integrated Environments

MetaCase MetaEdit+[24] is a commercial integrated environment for MDE. It provides a collection of tools to design a DSML by defining the concepts, constraints associated to the concepts, symbols representing the concepts and a generator to produce the desired software artefacts with *MetaEdit+ Workbench*. Modelling using the defined DSML is hosted within the MetaEdit+ environment (see Figure 3.16). The *MetaEdit+ Model* extends the generic modelling environment to provide a range of different views allowing the development team to view and edit or manipulate the model as a diagram, matrix or table. In addition, it provides the necessary tools for management of the model and its contents and provides support for a multi-user environment.

[24] Read more about MetaEdit+ from http://www.metacase.com/.

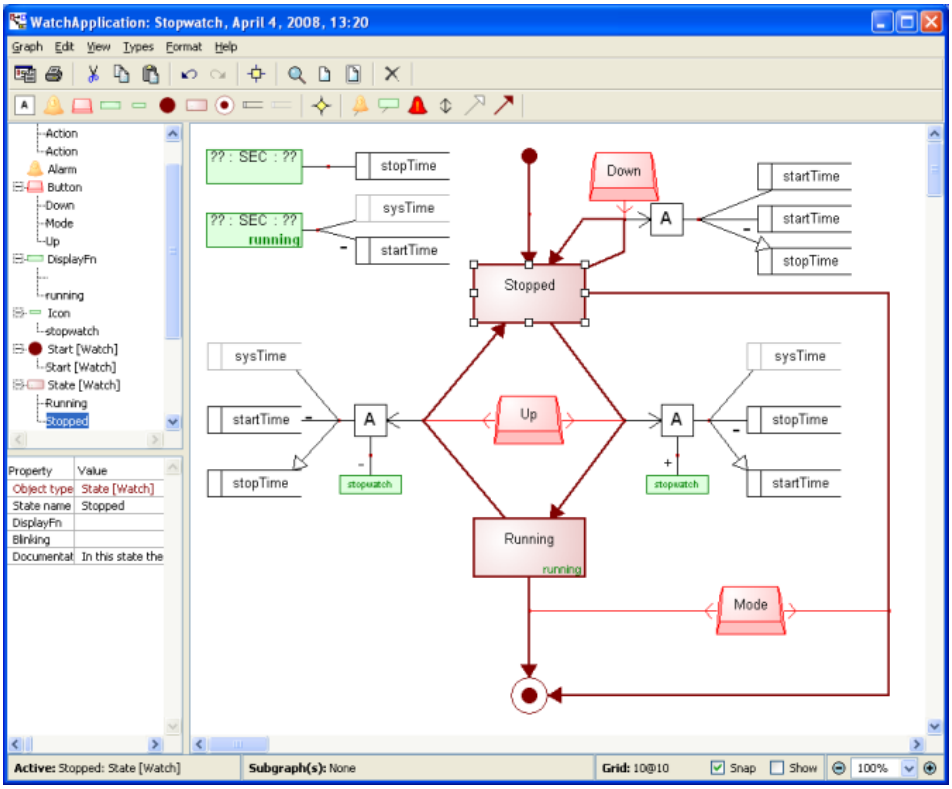

Figure 3.16: Screenshot of MetaEdit+ Workbench Diagram Editor
(http://www.metacase.com/mep/diagram_editor.html)

Microsoft DSL Tools is another integrated environment for MDE made available as an extension to the *Microsoft Visual Studio IDE* (S. Cook, et al., 2007). It provides the necessary tools for defining a DSML, modeller and generator to generate textual artefacts from the model within the Visual Studio environment (see Figure 3.17). In DSL Tools, a domain model is created using a UML-like modelling language and a DSML is then represented with graphical notations consisting of shapes and connectors. Relevant generators can be defined in *Extensible StyleSheet Language Transformation* (XSLT) and text artefacts are generated through the Microsoft XSLT engine or by through a parameterised text template generation approach.

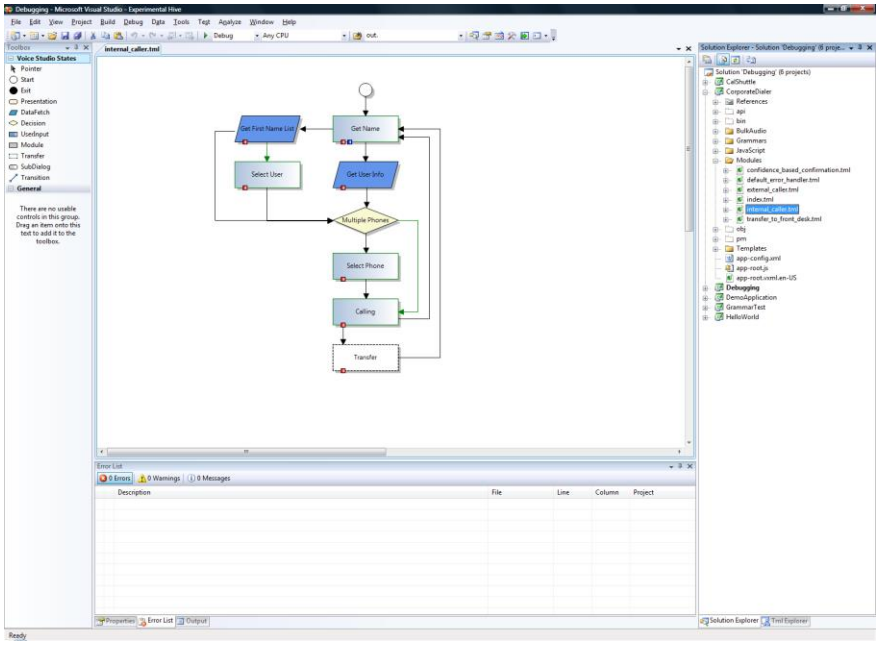

Figure 3.17: Screenshot of Microsoft DSL Tools
(http://blogs.msdn.com/garethj/archive/2009/02/06/tellme-voice-studio-beta1.aspx)

Visual Paradigm's Smart Development Environment[25] (SDE) is another commercially developed plug-in to support MDE practices which offers capabilities to model in UML, entity relationship diagram, data flow diagram and others, generate code for 15 different programming languages, team collaboration capabilities, round-trip engineering and can be integrated with well-known IDEs such as Visual Studio, NetBeans, Eclispe and intelliJ IDEA. The modelling environment is hosted within the supported IDE and code is instantly generated in the code editor view allowing both forward and reverse engineering.

Apart from commercially driven integrated environments for MDE, there are also non-commercial products available. GME, developed at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, provides a similar facility for defining a DSML and hosting modelling within the same environment (Ledeczi et al., 2001) (see Figure 3.18). In GME, a domain model is modelled in UML and assigned with graphical representation while modelling rules are defined using *Object Constraint Language* (OCL). Models are then saved directly in XML format or translated using an external interpreter (generator) into other textual artefacts using *Microsoft Common Object Model* (COM) technology through extensions to the GME.

[25] Read more about SBE from http://www.visual-paradigm.com/product/sde/.

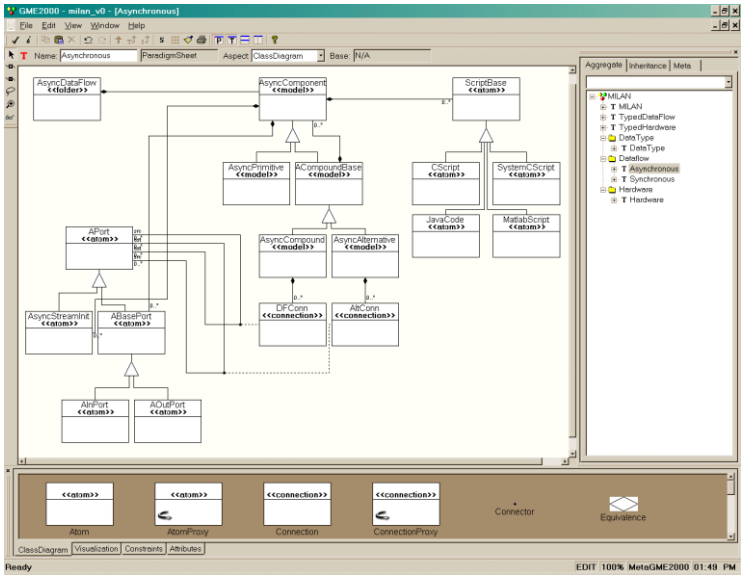

Figure 3.18: Meta-modelling in GME (Ledeczi, et al., 2001)

GEMS[26] is a meta-modelling tool developed by ISIS at Vanderbilt University that provides GME-like meta-modelling to create modelling tools that can be hosted within the Eclipse environment. It uses Eclipse modelling technologies such as EMF and *Graphical Modelling Framework* (GMF). Like *GME*, a DSML is created in *GEMS* using UML-like notations and later ported automatically to Eclipse as plug-in. Generators and transformation engines can be created for the modelling environment created using *Open Architecture Ware*[27] (OAW), *Java Emitter Templates*[28] (JET), and *Atlas Transformation Language*[29] (ATL) to transform the model into desired textual artefacts.

The *NetBeans IDE*[30] also provides similar facilities for modelling using a *UML plug-in*[31] and code generation using a template engine such as *FreeMarker*[32]. The *UML plug-in* enables the modelling environment to model the SUS using a range of supported UML diagrams such as activity diagram, class diagram, sequence diagram, state diagram, and use case diagram as a Java platform model or PIM. The *FreeMarker* code templates associated with the diagrams can then be customised to generate the desired Java source code.

Among the reviewed integrated environments for MDE, MetaCase MetaEdit+ offers comprehensive and customisable modelling features, which can be integrated into many IDEs, and it offers a facility to create generators for any text artefacts. GME is also a standalone tool with a similar

[26] Read more about GEMS from http://www.eclipse.org/gmt/gems/.
[27] Read more about OAW from http://www.eclipse.org/workinggroups/oaw/.
[28] Read more about JET from http://www.eclipse.org/modeling/m2t/?project=jet.
[29] Read more about ATL from http://www.eclipse.org/m2m/atl/.
[30] Read more about NetBeans IDE from http://netbeans.org/.
[31] Read more about UML Plugin for NetBeans IDE from http://netbeans.org/features/uml/.
[32] Read more about FreeMarker from http://freemarker.sourceforge.net/.

aim, but suffers from the lack of a built in generator. For Visual Studio users, Microsoft DSL Tools is an alternative to MetaEdit+ where a DSML can be created, hosted and generate code entirely within the Visual Studio environment. Similarly, GEMS provides a facility to create the modelling environment as an add-on to the Eclipse IDE with support to create generators for any text artefacts. Each of the reviewed integrated environments provides the facility to support the MDE process in one way or the other. The SDE allows users to model using UML diagrams, entity relationships diagram, BPMN, process map and others. It also provides great support for code generation and integration with well-known IDEs. The NetBeans IDE with the UML plug-in is an open source alternative to SDE. Selection of the integrated environment will ultimately depend on the available budget to invest and choice of development tools. These MDE integrated environments can be used for development of a model-driven game development environment similar to those proposed in Section 3.7 and realisation of a model-driven game development environment tailored for non-technical domain experts will require a model-driven framework that supports development of computer games similar to those described in Section 3.6.

3.8.2 Code Frameworks

Code frameworks for MDE can generally be divided into two categories; *diagramming* and *transformation*. IBM's *ILOG Visualisation*[33] and the *Eclipse Graph Editing Framework*[34] (GEF) are examples of diagramming code frameworks that provide the software components such as 2D graphics, notation palette, environment layout support, viewer, connection anchoring, connection routing, connection decoration and cursor support which are necessary for creation of a diagramming environment. ILOG Visualisation is licensable as GUI components for the .NET, Java, C++ and Adobe Flex platforms, whereas GEF is only available as a plug-in on the Java platform specifically for the Eclipse IDE.

The *Eclipse Modelling Framework* (EMF), *CodeWorker*[35] and ATL are examples of transformation code frameworks. The EMF provides the facility to build a code generator in the Eclipse IDE. It accepts a model expressed in annotated Java, XML or *XML Metadata Interchange* (XMI) and transforms it into Java implementation classes or a formatted XML document. An alternative to EMF is CodeWorker which is freely available as a Java interface, .NET assembly and an Eclipse plug-in. For

[33] Read more about IBM ILOG Visualisation from http://www-01.ibm.com/software/websphere/products/visualization/.
[34] Read more about GEF from http://www.eclipse.org/gef/.
[35] Read more about CodeWorker from http://codeworker.free.fr/.

model-to-model transformation, the ATL can be used to transform a set of source models to a set of target models. The ATL is offered as plug-in to the Eclipse IDE.

The Eclipse *Graphical Modelling Framework*[36] (GMF) provides a full-featured modelling environment by integrating the diagramming features from GEF and the transformation features from EMF. This reduces the complication for developers who want to create a full-featured modelling environment in the Eclipse IDE. For non-Eclipse users, modelling environments can be built on the integration of ILOG Visualisation and CodeWorker. Building MDE tools based on these code frameworks described can be time consuming compared to the use of an integrated environment described in Section 3.8.1. However, it provides great flexibility for developers to create custom interfaces and target code generation to required platforms more accurately.

3.9 Assistive User Interfaces

The game modelling environments reviewed in Section 3.7 use a range of user interfaces to capture game requirements from users. Diagramming interfaces that use shapes (or images) and connectors, wizards, visual editors and text editors are standard approaches found in game modelling environments. Each of these approaches provides a different form of support to users in defining the game world as some require a visualisation environment and some do not. Selection of the right interface is crucial especially in the view of this research study as the primary users are domain experts who may not have much technical knowledge in games development.

Assistive user interfaces provide guidance to users in preparing a model. Software based 'wizard' interfaces are a simple way to guide users through systematic information entry. However, they require a number of custom 'wizard' interfaces to enable users to enter details of a model during modelling and do not provide a high-level view of the model. Diagramming interfaces and visual editors provide some form of guidance through visual cues and feedback from interactions. They do, however, require users to undergo some form of training or tutorial before they become proficient in using the tool. Among all user interfaces, the text editor provides the least guidance in entering formalised information as it requires statements to be entered that conform to the syntax of some formal language. Although modern text editors include syntax colouring, code tooltips, error highlighting and many other advanced facilities built-in, users are still prone to committing errors syntactically.

[36] Read more about GMF from http://www.eclipse.org/modeling/gmf/.

Alice (Kelleher et al., 2002) and Scratch (Maloney et al., 2004) are examples of applications that use assistive user interfaces developed to address the needs of non-technical users. Alice is an educational software development tool that motivates and aids students in learning computer programming within a 3D environment. Understanding the frustration of students learning programming, Alice 2.2 and Alice Storytelling are based on a drag and drop interface that addresses the major problem of many learners i.e. syntax errors (Kelleher, et al., 2002). Commands, programming constructs, 3D objects, object properties, and behaviour are represented as tiles with different colours which users can drag into the animation and behaviour areas with a menu that displays the available options for selection of parameters to complete a statement that represents the animation or behaviour of an object in a 3D world (see Figure 3.19).

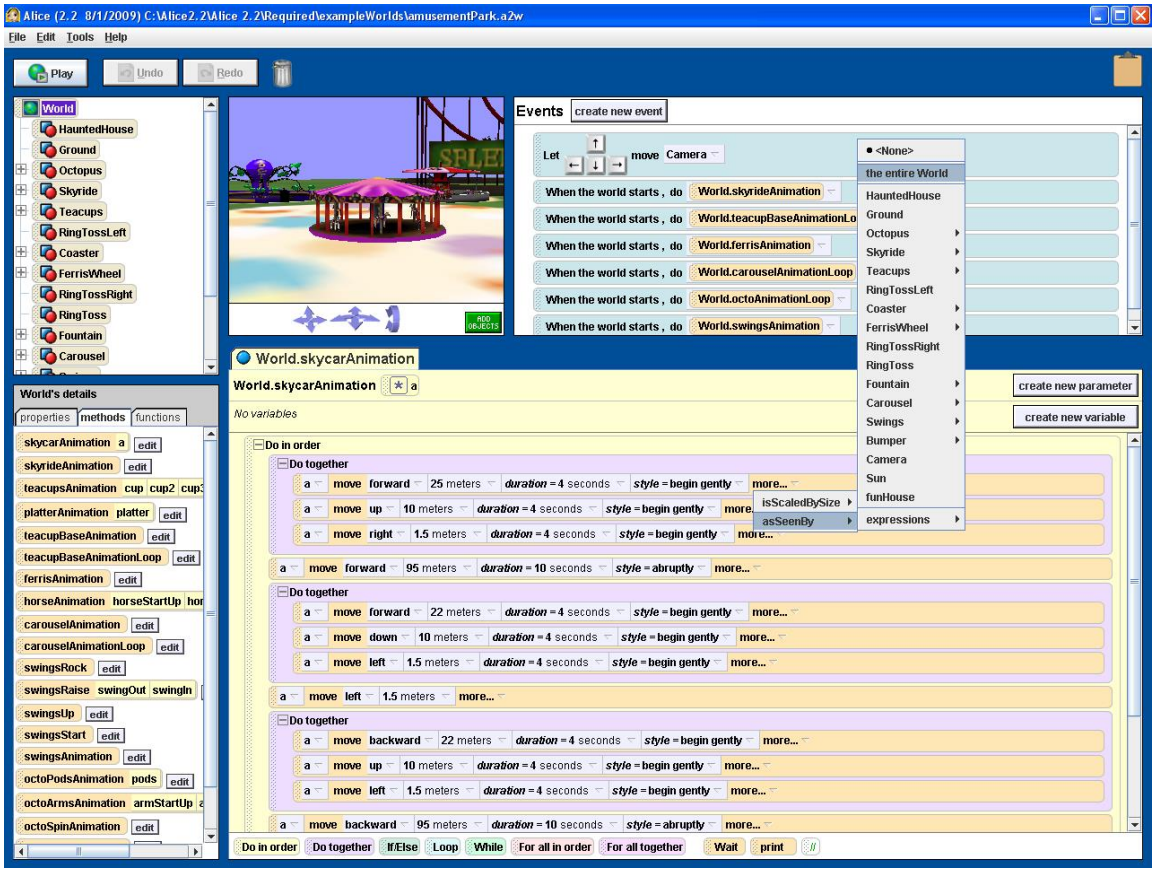

Figure 3.19: Screenshot of Alice 2.2 User Interface

Scratch is another initiative similar to Alice developed by the MIT Media Lab to assist children learning programming concepts (Maloney, et al., 2004) (see Figure 3.20). It provides a rich media authoring environment with much simpler interfaces and uses a graphical programming language to enable learners to script multimedia objects and interactivity. Programming constructs are grouped into motions, controls, looks, sensing, sound, operators, pen and variables using puzzle-style coloured

block as notations. Within each construct, values can be inserted via the textboxes or list boxes available as shown in Figure 3.20. Blocks are fitted together based on the compatibility of the command and data shapes to eliminate syntax error, thus lowering the barrier of learning programming for beginners.

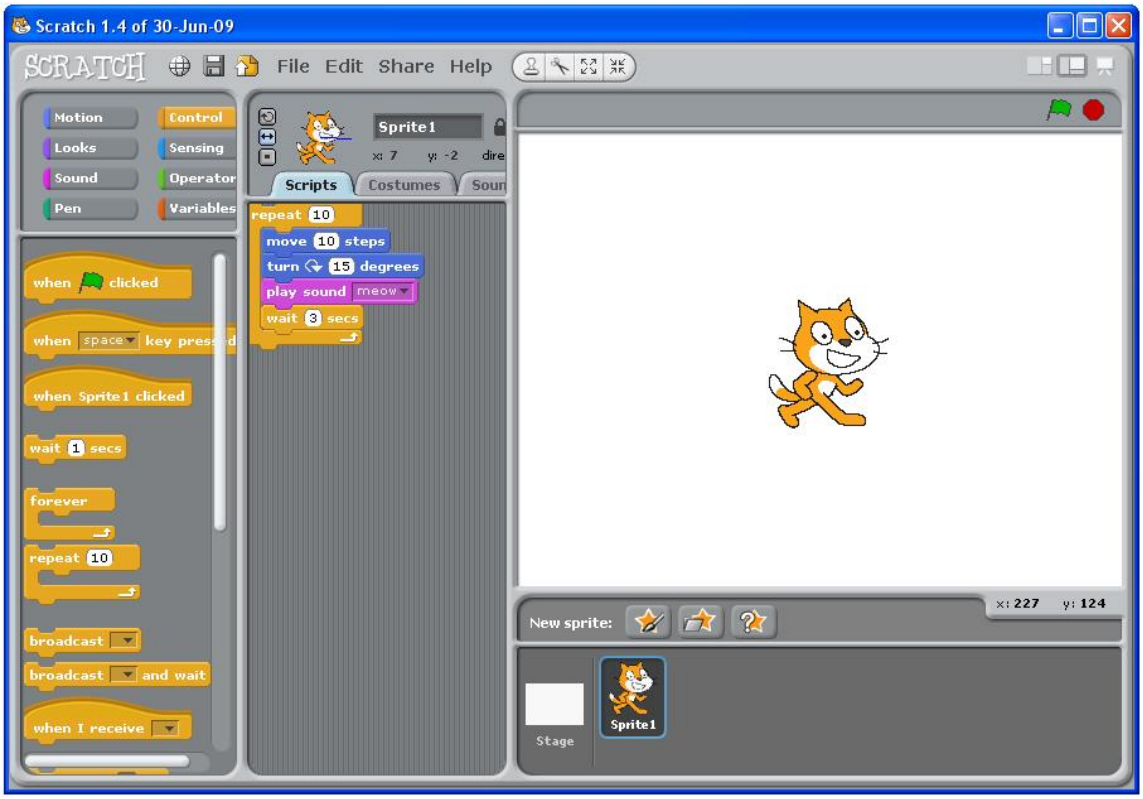

Figure 3.20: Screenshot of Scratch User Interface

Both the reviewed user interfaces are examples of block-styled drag-and-drop interfaces with clever use of input type to simplify interactions. The constraints introduced in these user interfaces eliminate unnecessary errors and also guide users in completing formalised text-entry. In general, such user interfaces can lower the learning curve when using technical tools and should be adapted into model-driven educational game development environments to aid non-technical domain experts in production of educational games.

3.10 Chapter Summary

This survey provides insights on developments in the area which plays an important role collectively on the creation of a model driven game development environment. We identified these approaches and solutions, and reviewed them from MDE perspectives. Although most of the current approaches and solutions are related to model driven game development, using these approaches and solutions

will require knowledge of the model at certain level and substantial modification before they can be deemed suitable to be integrated in a model driven game development framework. Nevertheless, it provides a reference for framework developers to begin with, extend or adapt. Although current efforts in model-driven game development mainly focus on mainstream audiences (i.e. industry and hobbyist), there is a growing awareness of the adoption of MDE specifically to support production of serious games particularly for education and training purposes.

CHAPTER 4 - REQUIREMENTS FOR A MODEL-DRIVEN GAME DEVELOPMENT FRAMEWORK

This chapter describes both the functional and operational requirements for the development of a model-driven framework to support computer games development. Since serious games is still a relatively new concept, most non-technical domain experts have very little knowledge of how games are designed and developed, and have to rely heavily on collaborative efforts with commercial game developers to produce appropriate serious games. As part of this research, the requirements for our model-driven framework are formulated based on our analysis on studies of games development and MDE to provide a comprehensive and collective view (Tang & Hanneghan, 2008, 2010a; Tang, Hanneghan, & El-Rhalibi, 2006, 2007; Tang, et al., 2009).

4.1 Functional Requirements

The functional requirements of the model-driven games development framework define how the framework should be designed to marry games design and games development into a cohesive contemporary domain to support the model-driven approach of games development. These requirements include embedding best practices for games design, supporting existing game software frameworks and generating artefact are discussed in the following subsections.

4.1.1 Support for Existing Game Software Frameworks

There are many game software frameworks available for use as described in Section 3.5. Each game software framework supports an individual list of features and offers a different combination of techniques to facilitate the development of computer games. It is a real challenge to support all games software frameworks in any model-driven framework as new games software frameworks are developed while existing ones get updated over time. Contrary, the framework should not be limited to support only on a selection of games software frameworks. Instead, the game software model in the model-driven framework should be abstractly designed to include the fundamental features and techniques that are required to produce serious games.

Extending the framework to support any game software framework would require an interface that can bridge between the model and the software artefacts. It is not feasible to 'plug' an unknown game software framework into the model-driven framework and expect it to automatically define relationships to the semantics of the game model and game software model. Such a matter must be resolved to allow extensibility of the model-driven framework to support as many game software frameworks as possible. This will be resolved in our model driven games development framework which will be presented in Chapter 5.

4.1.2 Generation of software artefact

Formal requirements in the form of a system model can be generated into a range of artefacts such as UML diagrams, XML documents, software code or even executable files. In a complex model-driven framework suitable for games development, generated artefacts must be linked with external media assets and game specific functionalities. In addition, artefact generation should be loosely coupled with the model-driven framework to provide flexibility for framework developers to develop custom generators.

4.2 Operational Requirements

Operational requirements define how the model-driven framework and the associated tools, assets and game functionalities should designed to fit in to the practices of domain experts. It is equally important as the aforementioned functional requirements to ensure the model-driven framework can assist domain experts to produce quality computer games easily and affordably. These requirements are discussed in the following subsections and include the game assets and functionalities, and user interfaces for a model-driven development environment.

4.2.1 Support for Externally Produced Art Assets and Game-Specific Functionalities

Computer games are more than just simulation software. They are also regarded as a form of digital art. It integrate game rules to dictate the interactions, game physics to simulate motion and physical behaviours, game AI to mimic decision making with audio and visual elements to enliven the game objects in the virtual world. Game rules can be specified by domain experts through declarative statements, but audio and visual elements will require specific skills to produce compatible technical assets with the game software framework using advanced media production tools. Similarly, defining physics equations and modelling artificial intelligence are technically demanding tasks. These

specialist art assets and game functionalities should ideally be outsourced to 2D artists, modellers, animators, sound designers and game programmers to ensure quality is not compromised.

To adhere to such a requirement, the game model and game software model must be designed to provide a reference point which can link to the externally produced elements for artefact generation. In addition, media will have to be available in the format acceptable by most game software frameworks. Game functionalities on the other hand will have to comply with certain guidelines to assure compatibility with the game software model. In order to have the game functionalities interoperate with other game software frameworks, specification of the game functionalities must be documented formally using the same format as the game software model to enable development of a suitable code generator.

4.2.2 User Interfaces in the Modelling Environment

User interfaces (UI) are crucial elements that encapsulate the technical aspects of modelling in the model-driven computer game development environment. It is the primary aspect that domain experts will encounter and which usability of the modelling environment will be judged on. While most MDE tools and MDD environments are developed for technical users, the model-driven computer games environment in this research study is meant for non-technical domain experts who may be unfamiliar with technical user interfaces. Based on the review of model-driven game development environments in Section 3.7, user interfaces must be simple, directed and non-technical. Therefore, user interfaces for modelling computer games should be designed based on a user-centred approach to address the needs of domain experts instead of game developers in order to:

- minimize unnecessary error,
- guide users in completing computer games design requirements formally, and
- encapsulate the technical aspects of games development.

4.3 Chapter Summary

In this chapter, we have identified three functional and three operational requirements for the development of a model-driven framework for computer games development. The functional requirements influence the design of the framework that guides domain experts and computer game designers in designing computer games using the model-driven approach, while the operational requirements define how the framework, tools, assets and game functionalities should work. We identified that the model-driven framework should be designed to support a variation of game

software frameworks in effort to provide domain experts with the options to pick and choose game functionalities and operating platforms that suit the requirement of computer game. Functionally, the model-driven framework should also be able to generate artefacts such as UML diagrams, XML documents, software code or even executable files. The framework should accept external art assets and game functionalities that are produced by professionals and these should be integrated into or linked with the generated artefact. Finally, we acknowledged the importance of simple, directed and non-technical UI-based modelling environments that encapsulate the technicality of computer games development from domain experts. These requirements will be the basis of our design decisions when designing our model-driven games development framework that is featured in Chapter 5.

CHAPTER 5 – A NEW MODEL-DRIVEN GAME DEVELOPMENT FRAMEWORK

Game-based learning can offer the aid much desired by domain experts by taking advantage of gaming technologies to create a new generation of educational technology tools that can better equip learners of all ages with necessary skills through such an innovative learning approach (FAS, 2006b). To address the need of high-level tools that can assist domain experts in creating games for game-based learning, findings from this research study reveal that Model Driven Engineering (MDE) can offer a basis for development of such high-level tools. This chapter unveils our model-driven game framework developed to this end. Our model-driven game framework defines the models and the relationships with other components such as user interfaces and MDE tools that form the basis for the building of model-driven games development environments.

The requirements outlined in Chapter 4 are guides for designing the ideal model-driven framework for supporting development of computer games. Such a framework will need interrelated but loosely coupled and well-defined components to ease translations of models, binding of externally developed assets and functionalities, and generation of software artefacts. These characteristics are also well exemplified in SharpLudus Game Software Factory (Furtado, 2006), MDGD framework (Reyno & Cubel, 2008) and Board Game Model (Altunbay, et al., 2009) reviewed in Chapter 3. The basis of a model-driven approach is to logically map the requirements to corresponding and technically elaborated semantics for composition of artefacts using models as the centrepiece of such a transformation. In addition to the requirements presented in Chapter 4, variability of games to be generated and interoperability of the games across different game software framework are factors affecting on the design of the model driven games framework.

5.1 Architectural Strategies for building a Model-Driven Games Development Framework

The three-level architecture approach to map requirements directly to corresponding codes described by Kelly & Tolvanen (2008) consists of a *domain modelling language*, a *generator* and a *supporting domain framework*. Requirements gathered during game design are represented as a game software

model and artefacts are generated for a specific game software framework using a code generator. This architecture is also seen in the MDGD framework and Board Game Model which are designed mainly for technical users. Similarly, it is used in the SharpLudus Game Software Factory which utilises a wizard-based user interface. The simplicity of this architecture tightly couples the game software model to the game software framework thereby limiting the type of computer game that can be produced. This solution still retains all the benefits of MDE described in Chapter 2.2 but is less appropriate for the context of this research due to the increased technical level of modelling required and lack of interoperability.

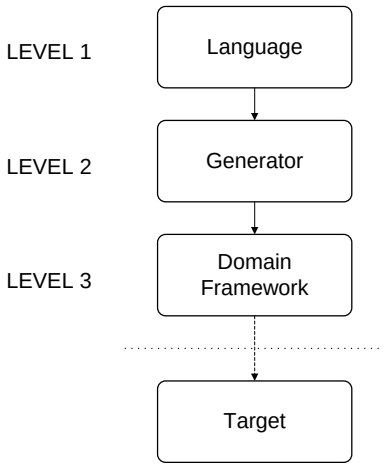

Figure 5.1: Three-level Architecture (S. Kelly & Tolvanen, 2008)

Tackling such issues would require a model that can encapsulate the technical aspects of computer games, relates well to domain experts, and supports a higher level of abstraction. The MDA (Miller & Mukerji, 2003) presents a potential solution for architecting an interoperable model-driven games framework of the aforementioned requirements. Unlike the three-level architecture which consists of only a single view of the software, MDA is more elaborate in the context of model representation. The model-level consists of three layers of models, namely a *Computation Independent Model* (CIM), a *Platform Independent Model* (PIM) and a *Platform Specific Model* (PSM) each of which offers progressively refined views with a different level of abstraction. CIM represents the logic of the SUS abstracted from the system structure, while the PIM refers to a computation-dependent model of the SUS but not tied to any hardware or software platform, whereas the PSM is computation-dependent and specific to a technology or language platform. These models are transformed from one view to another using special transformation tools before finally being generated into the software code for a targeted platform. It is obvious that MDA offers a framework setup with higher level of abstractions

which allows these models (CIM and PIM) to be reused and different resultant models (PSM) to be generated via alteration of transformation rules.

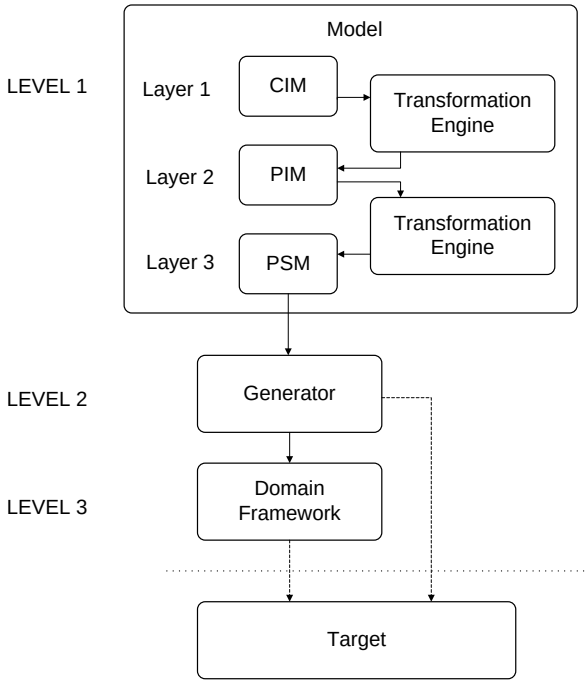

Figure 5.2: OMG's MDA

From a computer games engineering perspective, the CIM represents the logical structure and building blocks of a computer game, while the PIM depicts the computer game as a software system independent of specific game technology representation. The PSM incorporates the required specific game technology details of a selected deployment platform to finalise representation of computer games. Game technology can exist in the form of game software frameworks or derived directly from multimedia APIs.

Computer games can be scripted via a scripting facility such as Lua (Manuel, 2002), coded using programming languages such as C and C++ or made with proprietary level data for a specific game software framework. It may seem that to achieve game software variability, the transitive relationship from PIM to PSM and generation of artefacts from PSM to target can be merged giving greater flexibility for game developers to have better control over the artefacts generated by directly mapping the game software model to artefacts in whichever way seems valid. The additional PSM provides a mechanism to support more target platforms without changing the underlying implementation. Framework specific computer game software can be modelled with more detailed care and attention by the framework developer resulting in a direct one-to-one specification to code mapping thereby simplifying the generation of software code.

5.2 The Model-Driven Games Development Framework

Our new model-driven game framework is featured in Figure 5.3 and consists of nine parts namely: (1) User Interface (UI), (2) Models, (3) MDE Tools, (4) Components Library, (5) Code Templates, (6) Artefacts, (7) Technology Platform, (8) Operating Platform and (9) Software. This configuration loosely couples the modules allowing the framework developer to flexibly substitute modules while maintaining the integrity of relationships among the modules via well-defined interfaces. It also clearly divides the views of entities while promoting structured and systematic workflow.

Encapsulating the models is the UI (1) module which can support input mechanisms such as natural language, script or even visual language, for example UML or flowchart, to specify the visual aspects of games. Separating the UI from the models enables the framework to be more accessible by different user groups, for example a wizard-based interface would suit non-technical users, natural language and visual notation might be more appropriate for intermediate users while script can be an option for advanced users. In this model-driven framework the UI represents the modelling environment while hiding the technical details of games development from domain experts. We believe that separating the UI elements from the model allows the framework developer to develop the right mix of assistive user-interfaces and relevant domain related vocabulary to guide non-technical domain experts to describe their computer games effectively.

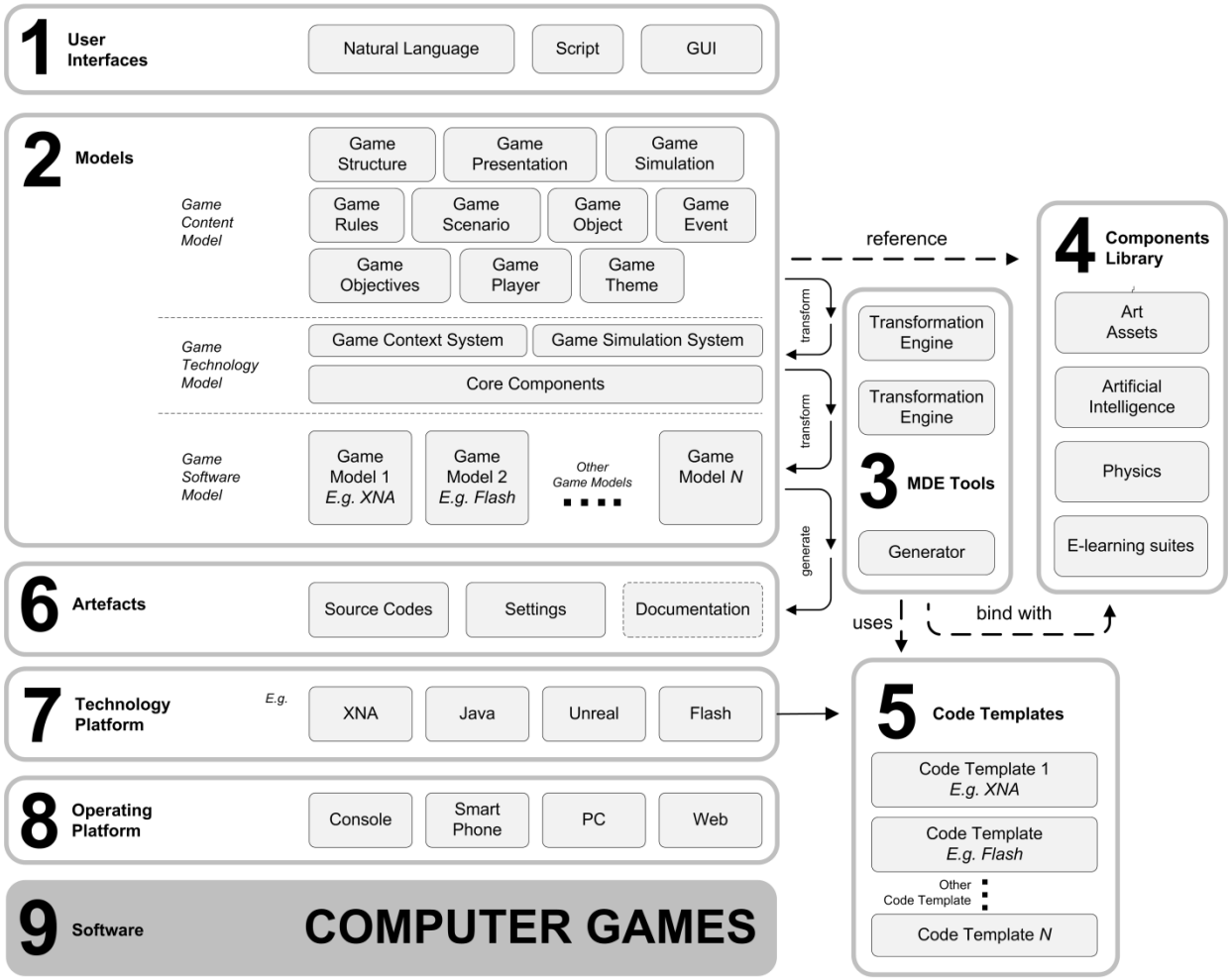

Figure 5.3: Model-driven Games Development Framework to support development of computer game.

At the core of this modelling framework is the *models* (2) module which represents a computer game in three different viewpoints namely:

- ***Game Content Model*** (GCM) - This represents the logical design specification of computer game as a model. The Game Content Model will be modelled by domain experts via a modelling facility defined in the UI module. It will be linked during modelling to indicate assets and other components used in the computer game and supported by the framework. A complete Game Content Model consists of models that represent the core aspects of computer games including definition of objects, their attributes, behaviours and linkages with art assets and game functionalities, events and progression, construction of a situation which consists of characters, objects, objectives, scripted events and problems to be solved through game-playing, tracking of interactions and various user interfaces for selection of game modes and display of information such as game objectives and results.

- ***Game Technology Model*** (GTM) is a computation-dependent model of computer games independent of technology or language platform. The Game Technology Model, mapped from the Game Content Model, models computer games from a software perspective representing the computer game in programmatic order and structure marked with additional and specific game-related functionality required by the computer game design. The aforementioned GameDSL is an example of Game Technology Model which can be adopted in this framework.
- ***Game Software Model*** (GSM) refers to the transformed model of the computer game specific to a technology platform.

By utilising this approach, it can be seen that a single game (Game Content Model) can be used to support a wide variety of platform and technology without change. Models are transformed from one viewpoint to another automatically. The Game Content Model is transformed to Game Technology Model and subsequently to Game Software Model and finally to artefacts using appropriate MDE tools. Models described can be represented in textual form such as eXtensible Markup Language (XML), or using graphical notations such as UML. Additional information is added to the Game Content Model during transformation to Game Technology Model and more game software framework specific information is automatically added to Game Software Model during the transformation. Transformation can be performed using specific MDE tools (3); a transformation engine can be used to mark models with additional information and generators can be used to export models into human-readable format (such as UML diagrams) or software code. The Game Technology Model transformation engine reads the Game Content Model and represents the game as a software model. The Game Technology Model is read by the Game Software Model transformation engine which reorganises the games as a software model compatible with a specific game software framework by replacing game logical software constructs with the corresponding physical game software constructs. Finally, the Game Software Model is interpreted by a generator to compose software code from predefined code templates (5) through mapping techniques. Game software framework constructs are defined by the framework developer collaboratively with developers of game software frameworks, while code templates are defined externally by framework developers by referencing the specification of Game Software Model.

Additional information is added to Game Content Model during transformation to Game Technology Model and more games software framework specific information is added to Game

Software Model during the transformation. Transformation can be performed using specific MDE tools; *transformation engine* can be used to mark models with additional information and *generators* can be used to export models into documents format (such as UML) which are reader-friendly or software code. The Game Technology Model transformation engine reads the Game Technology Model and represents computer games as a software model. The Game Technology Model is read by Game Software Model transformation engine which reorganises computer games as a software model compatible to a specific game software framework by replacing game software constructs with the corresponding game software constructs. Finally, the Game Software Model is interpreted by the generator to compose software code from predefined code templates through mapping techniques. Game software framework constructs are defined by framework developer collaboratively with the developer of game software framework, while code templates are defined externally by framework developer by referencing the specification of Game Software Model (see Figure 5.3 and Figure 5.4).

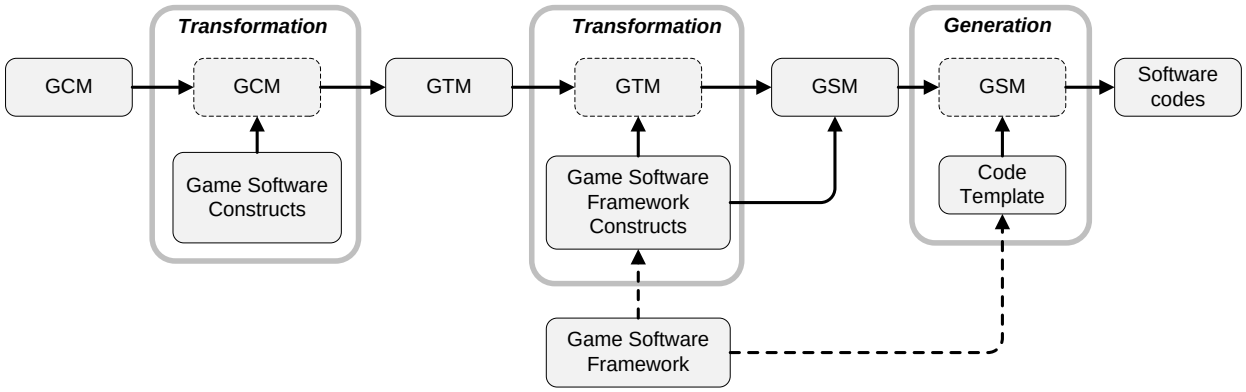

Figure 5.4: Transformation pipeline.

The technology platform (7) and operating platform (8) modules are developed externally by their respective parties. The configuration of this model-driven framework enables the support for a wide range of game software frameworks available in the market from open source to commercial promoting interoperability and variability of computer games.

5.3 Game Content Model (GCM)

Our Game Content Model is developed based on our studies and understanding of game design and development. It represents a game ontology from an interactive content viewpoint. It will be used to document the design specification of a computer game and will be the model for building other game models in our model-driven games development framework. The current scope of our game ontology

covers the game concepts used in the documentation of role-playing and simulation game genres which consist of concepts sufficient for modelling most casual computer games and serious games.

The development of our Game Content Model follows a bottom-up approach which requires us to identify the aspects of game design by first investigating game representation from a programmatic perspective before identifying the necessary elements that made up our model. This approach allows us to ensure that each concepts introduced in our model can be programmatically represented and it also helps us to identify those common game design that are not explicitly represented in the software. This also allows us to encapsulate the technical aspects of game development from game designers. This sets our ontology apart from GOP which was only useful for the purpose of game studies whereas the ontology we proposed in this article is ready for used in MDE projects.

The Game Content Model improves on the work of GOP (Zagal, et al., 2005), RAM (Järvinen, 2007) and NESI (Sarinho & Apolinário, 2008). We selectively combined these with our studies on game design, game development and serious game and organise the concepts in a meaningful object oriented structure which helps us to translate using MDE tools later. The top-level of our Game Content Model consists of four key concepts that best represent the rules, play and aesthetic information of a computer game and they are Structure, Object, Scenario and Mechanics. These key concepts are further decomposed into ten specific concepts that define a computer game and they are *Game Structure, Game Presentation, Game Simulation, Game Object, Game Theme, Game Scenario, Game Event, Game Objective, Game Rules and Game Player*.

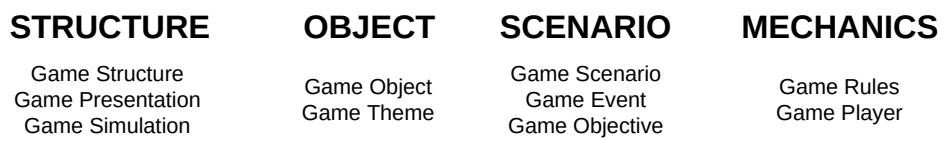

Figure 5.5: Concepts in Game Content Model.

In brief, the game structure provides the form and organises game into segments of linked game presentations and game simulations. The interactions between game object and the results of an interaction in a game simulation are defined using game rules. A game simulation can be used to host multiple game scenarios aligned with the storyline. Each game scenario is set up using a selection of game object to create an environment, a sequence of game event and a set of game objective that challenges game player skills and knowledge about the game. The Game player can control game object(s) and interact with other game object(s) via hardware or graphical user interfaces. And finally,

the game theme describes the "look and feel" of the game. At the highest level, the definition of a computer game is composed of a title, author's name, a game structure and one or more game players. This Game Content Model is also made available in Backus-Naur Form (BNF) in APPENDIX A: Ontology for Game Content Model. The relationships between these key concepts are illustrated in Figure 5.6.

The remainder of this section elaborates these ten specific concepts using relevant examples. These will be described in order of Game Structure (Section 5.3.1), Game Presentation (Section 5.3.2), Game Simulation (Section 5.3.3), Game Object (see Section 5.3.4), Game Scenario (Section 5.3.5), Game Event (Section 5.3.6), Game Objective (Section 5.3.7), Game Rules (Section 5.3.8) and Game Player (Section 5.3.9), Game Theme (Section 5.3.10).

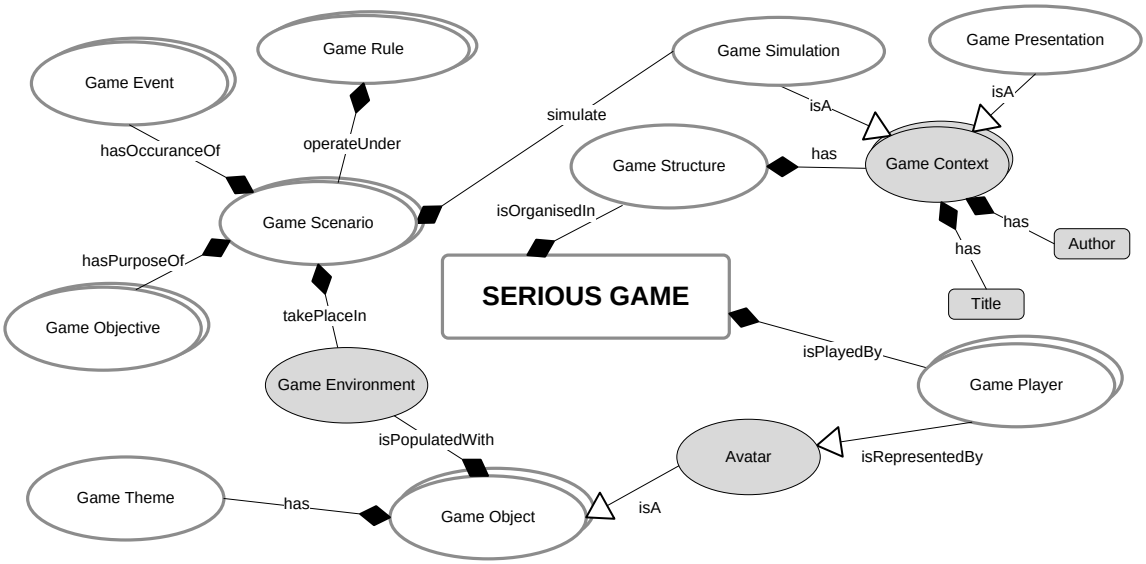

Figure 5.6: Overview of Game Content Model.

5.3.1 Game Structure

The game structure, as explained in the previous section, describes the architecture and the flow of a game. In our review of game design languages, we found that both Rich Pictures (Tang, et al., 2004) and Flowboard (Adams, 2004) can be used to model a game structure. However, they are less suitable for MDE purposes due to lack of formalism in describing states and transitional condition. Instead, we are adapting the Finite-State Machine (FSM) to accommodate our requirements in modelling game structure. In our ontology, we denote state as a *game context* (see Section 5.3.1.1). A game structure consists of a collection of game contexts which are linked through the definition of event triggers or GUI triggers (see Figure 5.7). A computer game can take the structure of complete game structure, scenario-based structure, training-based structure and presentation-based structure as described in

Section 4.1.1.2 depending on the type of instructional purpose. A full BNF representation of Game Structure is available in Table A.1.

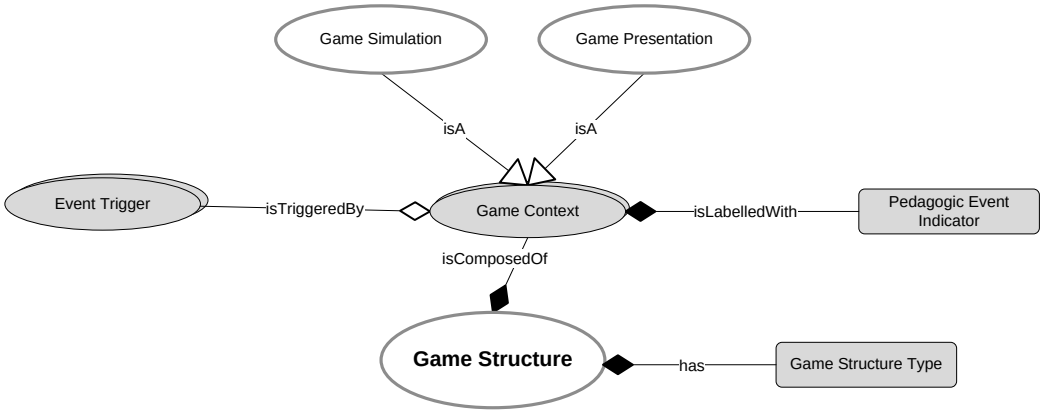

Figure 5.7: Ontology Diagram for Game Structure.

5.3.1.1 Game Context

A game context describes the type of game content to be presented to game players. In our game ontology, the game context can be either in the form of *game presentation* or *game simulation*. A game presentation is used to present information about the game using media and GUI components, whereas the game simulation is the actual game that consists of interactive contents that demands participation of the game players to achieve a set of defined goals in order to win the game or proceed to the next level of the game. Further descriptions of game presentation and game simulation are detailed in Section 5.3.2 and Section 5.3.3 respectively. Each game context also consists of a set of *event triggers* (see Section 5.3.1.3) which specifies the next game context to be presented when a transitional condition of a trigger is met. Additionally, the pedagogic event description (see Section 5.3.1.2) labels each context to indicate type of learning events associated to the game context.

5.3.1.2 Pedagogic Event Descriptor

Computer games designed for use in education and training or computer games must include some form of pedagogy. In this model, each game context is associated with one or more pedagogic events from the Gagne's (1970) nine events of instructions. These events ordered in sequence are (1) gaining attention, (2) informing learning objectives, (3) recalling prior learning, (4) presenting learning content, (5) providing learning guidance, (6) eliciting performance, (7) provide feedback, (8) assess performance and, (9) enhance retention and transfer. Labelling each game section with these pedagogic events invite game designers to think critically about and reflect on the learning contents

and activities that they should embed within the game match the type of instructional events. The requirement to include all nine pedagogic events within a computer game depends on usage of it in the context of game-based learning. A computer game can be designed for individual learning would need to meet all nine instructional events. Whereas a teacher who want to use a game as part of an exercise in a lesson may only include the pedagogic events "providing learning guidance", "eliciting performance" and "providing feedback", while the rest of the pedagogic events are delivered traditionally via lecture and coursework component. This will be used as a mechanism to check if a game has been designed for a full or a partial learning experience.

5.3.1.3 Event Trigger

Event triggers are used to invoke the transition between game sections and activate the game events or mark game objective within a game scenario. They can be classified into four distinct classes, namely:

- *Input trigger* which detects user input via hardware interface or graphical user interface (GUI);
- *Time trigger* which is essentially a countdown timer with an interval value with frequency of occurrence;
- *Proximity trigger* which behaves like input trigger but has a hotspot that detects the collision of a specified game object, a class of game object or a group of game objects from different classes; finally,
- *Game mechanics trigger* which is associated with a range of game application-related events such the media event (example is "onMediaEnd") and the simulation events (examples of simulation event "onSimulationEnd", "onSimulationPause" and "onObjectiveUpdate").

Each event trigger is associated with a command that specifies the transition between game contexts or an activation of a game event (see Figure 5.8).

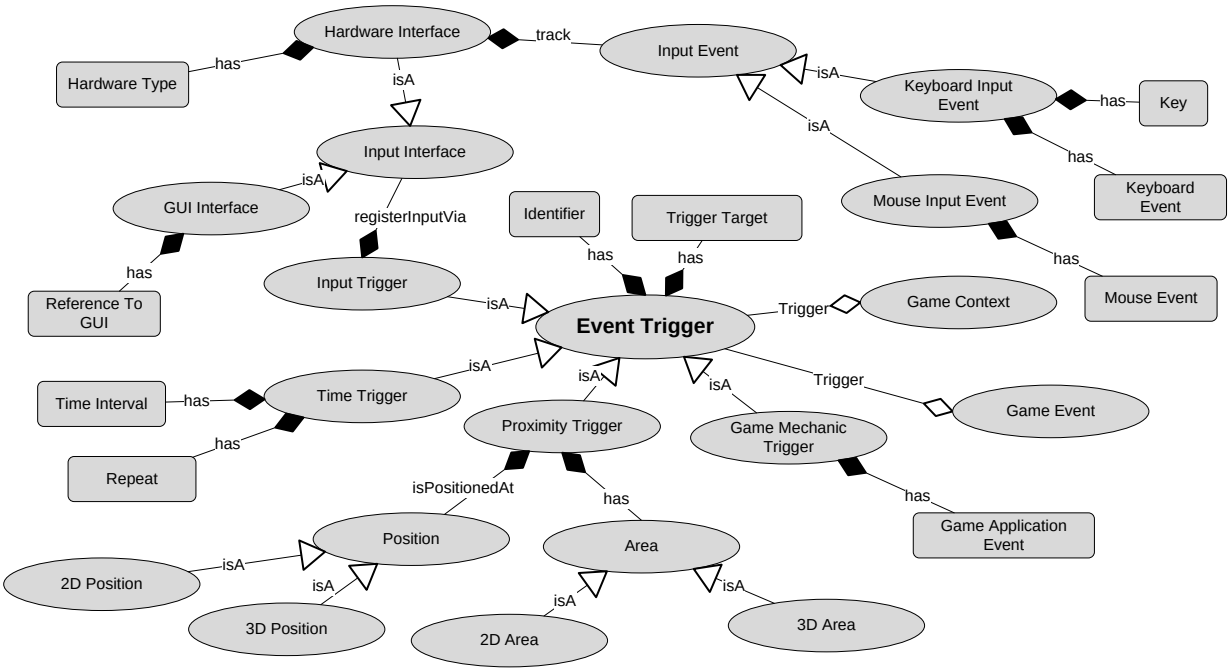

Figure 5.8: Ontology Diagram for Event Trigger.

5.3.2 Game Presentation

A game presentation is a virtual canvas that holds *media components* and *GUI components* to form a *game menu*, a *game notification* or a *cut-scene* to present information about the game and allow game player to navigate through the game structure. As a virtual canvas, a game presentation has properties such as depth, dimension (height and width) and even coordinate (x and y) on the device screen (see Figure 5.9).

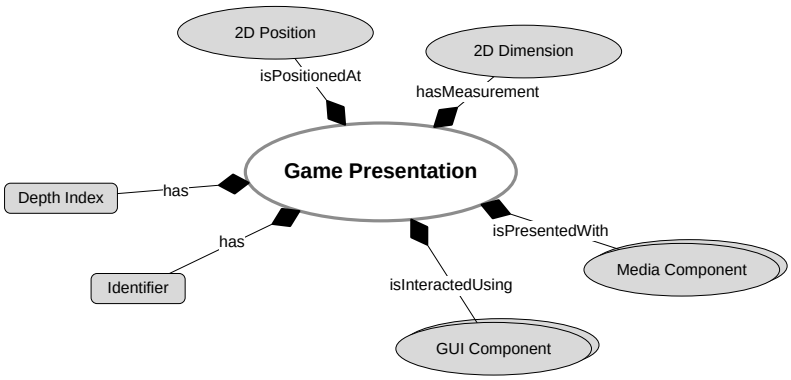

Figure 5.9: Ontology Diagram for Game Presentation.

A typical game menu presentation is composed of a virtual canvas that holds a background image or animation, GUI component that link to other game sections and possibly a sound loop. An example of game menu presentation for WarCraft III (Blizzard Entertainment, 2002) is shown in Figure 5.10. In games such as Need for Speed: Shift (Slightly Mad Studios, 2009), the game menu

presentation is slightly more sophisticated where the visual of the car and its custom upgrades are immediately reflected on the screen (see Figure 5.11). No matter how fancy the user interface is, fundamentally a user's input is mapped to a functional aspect of the game via a custom GUI component.

Figure 5.10: Menu in WarCraft III. (Screenshot: http://us.blizzard.com/support/image.html?locale=en_US&id=226).

Figure 5.11: Need for Speed Shift visual menu. (Screenshot: http://need-for-speed-shift.blogs.gamerzines.com/files/2009/08/paintmonstrosity.jpg).

A game notification presentation can have a background image with texts, more graphics, and button overlaid on the background image. It is used mainly to brief the game player about the game objective before the game simulation, to prompt player about in-game happenings during the game

simulation and to present to the game player the results after a game simulation (See definition of Game Simulation in Section 5.3.2). The virtual canvas for a game notification can be varied in dimension and can be displayed as an overlay on another virtual canvas. An example of game notification overlaid on the game simulation is seen in the Darfur is Dying (Susanna & Take Action games, 2009) serious game (see Figure 5.12).

Figure 5.12: Game notification in Darfur is Dying. (Screenshot: http://www.darfurisdying.com).

A cut-scene presentation usually consists of only a pre-rendered video that fills the entire screen. It has no GUI components and often the method use to skip the video is by pressing any key on the hardware interface. Cut-scene presentation is used widely as a platform for game storytelling. Alternatively, game story can be told using graphic with animated texts and sound. Modern computer games such as Red Dead Redemption (Rockstar San Diego, 2010) and KillZone 3 (Guerrilla Games, 2011) uses scripted real-time animation to convey information taking advantage of the game engine and to present a seamless transition between storytelling and game-play. A full BNF representation of Game Presentation is available in Table A.2.

5.3.2.1 Media Component

The types of media component supported in a game include *text*, *graphic/image*, *sound* and *video*. Each media component loads its content from a defined *media source*. For visual media such as text, graphic and video, additional properties such as coordinate (x and y) and dimension (height and width) are required (see Figure 5.13).

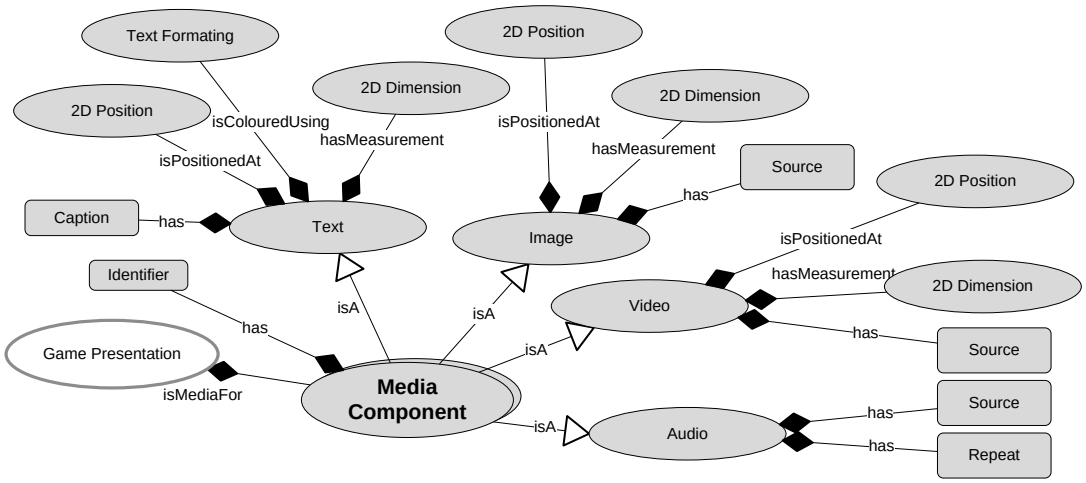

Figure 5.13: Ontology Diagram for Media Component.

5.3.2.2 GUI Component

Button, list box, check box, radio button and text box are examples of GUI components used in most software applications. Each GUI component has properties such as position (x and y), value and visual representation which can be styled accordingly to suit the game theme (see Figure 5.14). In the Game Content Model, designers will only have to provide necessary information to describe the components whereas technical issues of UI such as text-overflowing and data-capture should be resolved by developers where solutions can be programmed in the generator or may have already been built into the game software framework. Response to an input event is specified separately using the event trigger (refer to Section 5.3.1.3) which is associated to a GUI component.

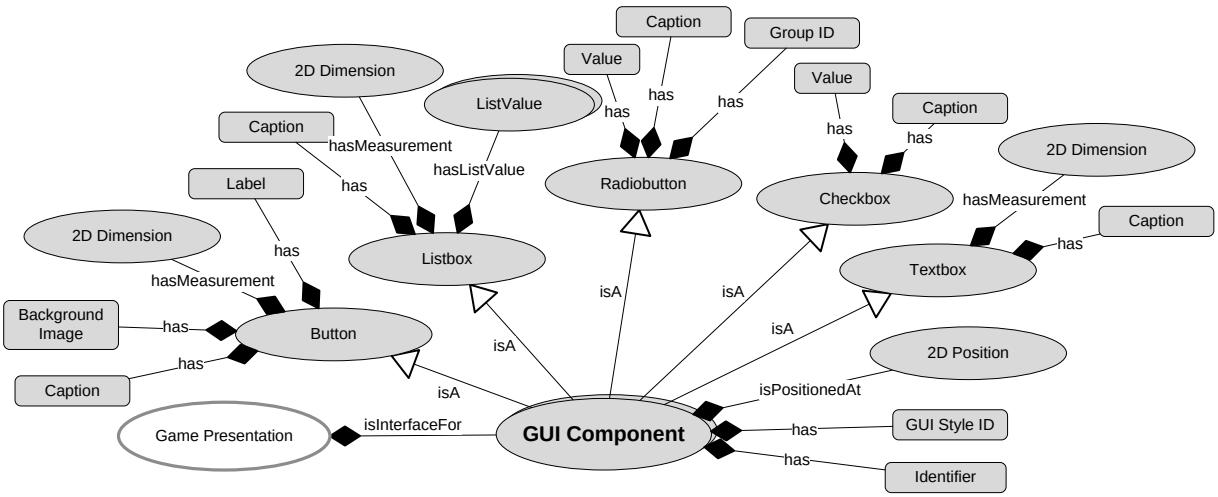

Figure 5.14: Ontology Diagram for GUI Component.

5.3.3 Game Simulation

Game simulation, by definition, is a mechanism that recreates scenarios virtually for game-play to take place. The simulation of a game scenario is governed by a set of rules that define the interactivity, and physical and temporal properties of the virtual world. A game simulation has *game rules* (read more about game rules from Section 5.3.8) *game dimension*, *game tempo* and *game physics* which give it a form (see Figure 5.15). *Game Scenario* (read more about Game Scenario from Section 5.3.5) is the content of the game simulation whereas *front end display* is used to display information about the simulation of a scenario to game players. In this section, we will be describing the concepts of game dimension, game tempo, game physics and front end display. A full BNF representation of Game Simulation is available in Table A.3.

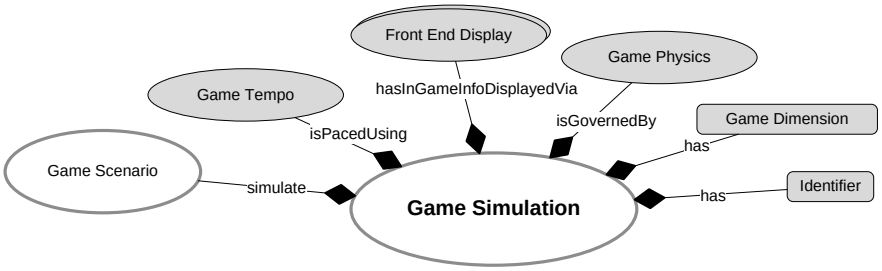

Figure 5.15: Ontology Diagram for Game Simulation.

5.3.3.1 Game Dimension

The game dimension refers to the virtual space of which the game simulation takes place. It can either be 2D or 3D. 2D limits the viewing of a game world to a side view (platformer or side scroller), top view or an angled view (isometric), whereas 3D provides the freedom to view the game world from all angles. Traditionally, the decision about the type of game dimension for a game has direct influence on the choice of game technologies and the production of graphic assets for use in the game. However, this is not the case anymore because a 2D game can be produced using 3D technology by fixing the camera to a plane or at a specific location. For example, Pacman is represented using 2D game dimension, whereas Halo Reach is rendered in 3D.

5.3.3.2 Game Tempo

Game tempo is the measure of pace of time in the game world. A minute in the game world may not refer to a minute in reality. For example, in soccer games such as Pro Evolution Soccer 2011 (Konami Computer Entertainment Tokyo, 2010) (see Figure 5.17), one half of the match which represents 45 minutes of the game world time can be set to represent 20 minutes of game-play in reality. In our

Game Content Model, it consists of a real-time which refers to the duration in reality and virtual time which refers to the duration in the virtual world (see Figure 5.16 for ontology on Game Tempo).

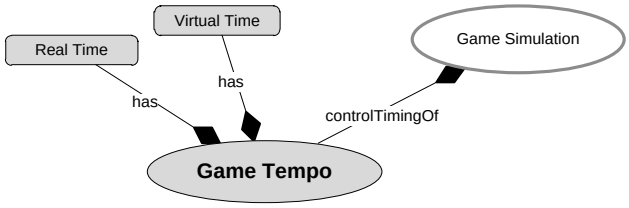

Figure 5.16: Ontology Diagram for Game Tempo.

Figure 5.17: Pro Evolution Soccer (PES) 2011 allows game player to adjust the game tempo to suit their desired game-play duration (Screenshot: http://game4us.net/wp-content/uploads/pes2011_3-140x140.jpg).

5.3.3.3 Game Physics

Game physics is used to define the physical state of the game world. It encompasses the *collision world* and *environment forces* (see Figure 5.18). The collision world, by default, is turned on to provide the illusion that game objects in the game world are solid. Turning off the collision world will cause the game objects to have a 'ghost' effect where solid object appearing to be going through another game object. There is no need for game designers to specify the low-level algorithms for collision detection as the focus in this model is to capture the specification of a game from a design perspective. The collision world will notify the respective game object and simulation if a collision has been detected and reaction to such an event could be an action from game object or to award a score based on the defined game interaction rule. What remains to specify in the game physics is the type of environmental forces that act on all game objects. Force is generally represented as a vector which has a value and a direction and can either be a constant or a dynamic value depending on how it is used to affect the game-play. An example of an environmental force in the Worms Forts: Under Siege (Team 17, 2004) (see Figure 5.19) is the wind speed of 3 meters and blowing in the north-west direction.

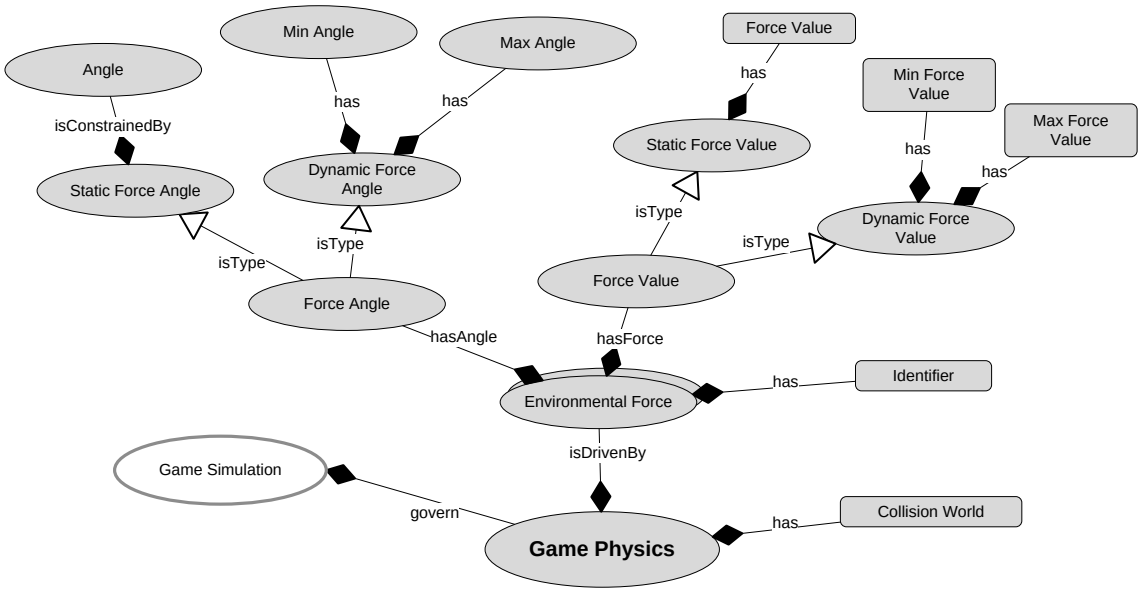

Figure 5.18: Ontology Diagram for Game Physic.

Figure 5.19: Worms Forts Under Siege uses the wind as one of the environmental force to affect trajectory of ammo. (Screenshot: http://www.wormsforts.com/images/mult/scre_22.jpg).

5.3.3.4 Front End Display

Front end display has no significant meaning to game design. It is just a mechanism to notify game players about the parameters in the game world that would affect their game-play. Front end displays are usually themed to provide a unified 'look and feel'. Some examples of front end display include:

- a *label* which uses characters to display textual information,

- a *counter* which uses digits or icons to display the value or number of an attribute,
- a *gauge* which is a recreation of an analogue display. It can be used to represent game time, speed of a game object or even used as a compass to represent direction,
- a *mini map* which provides the location of specific game objects within a certain proximity, and
- a *bar* which is another visualisation method to indicate the remaining value of an attribute.

Each front end display would also require information about the data source and the position of it on the screen. The position of the front end display can be fixed on the screen or be relative to the position of the game object (see Figure 5.20). In computer games such as PES 2011 (Konami Computer Entertainment Tokyo, 2010), static front end displays are used to present information such as a score for each team, remaining time and a mini map of the players in the field as seen in Figure 5.17. Another example is Halo: Reach (Bungie, 2010) which uses a mix of static and dynamic front end displays to present not only information about the game scenario, but also the ammunition level of the selected weapon and other non-handheld weapon such as grenade (see Figure 5.21). These front end displays have fixed position and are displayed when game objects are selected. An example of relatively positioned front end display is seen in the computer game Worms Forts: Under Siege (Team 17, 2004) in which it is used to display the name of each "worm" and their health level (see Figure 5.19). In the game simulation concept, we only need to specify the description of all static front end displays and the position of any dynamic front end display so to maintain consistency of layout throughout the game. The details of relatively positioned dynamic front end displays will be described through the game object itself as an internal representation (see Section 5.3.2.4).

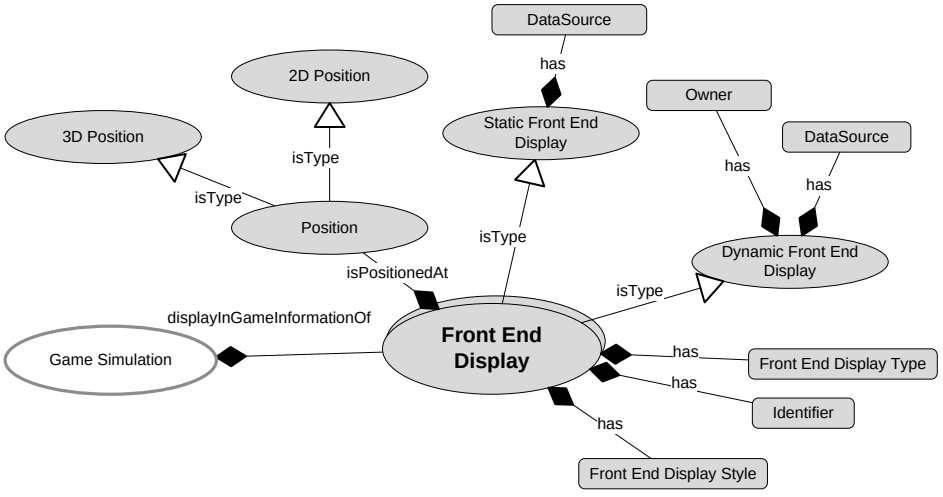

Figure 5.20: Ontology Diagram for Front End Display.

Figure 5.21: Halo Reach uses both static and dynamic front end display to provide the necessary game statistics to game player. (Screenshot: http://img94.imageshack.us/img94/3760/avermediacenter20100906.jpg).

5.3.4 Game Objects

Game objects are virtual things that populate the game world which can be designed to combine abilities such as decision-making, moving, acting and responding to surroundings and the game player's input simulating their existence in the game world. From a software design perspective, Object Orientation (OO) is the best approach to model objects. It invites practitioners to think about their characteristics and behaviours. However, OO is only a conceptual guiding definition of relevant characteristics and behaviours for representing the complexity of game object. It does not provide any structure which can aid a non-technical domain expert to define any game objects. Similarly, GOP (Zagal, et al., 2005) is also inappropriate for modelling objects as it generally describe object as an ontology entry rather than specifying what it represents.

In our Game Content Model, we adapted the Viable System Model (VSM) (Espejo, 1990), a conceptual tool to understand organizational structure, as a model to define game objects. The VSM is modelled after the human nervous system and classical cybernetics theory to represent an organization and we happen to find it is suitable for use in modelling game object because what it was modelled after. The only drawback of using VSM to model game object is its bureaucratic nature. However, this is not the case for game object definition. Our application of VSM is merely a model for defining game objects as opposed to modelling a software object itself. Our application of VSM as a

game object model is illustrated in Figure 5.22. The VSM representation of game object is described as follows:

- System 5 (*Personality Manager*) represents the method that initialises the in-game component based on some defined personality.
- System 4 (*Sensor Manager*) represents methods related to collision detection which are triggered at appropriate intervals.
- System 3 (*Action Generator*) can be considered as a set of pre-defined methods for updating vital data, action states and AI-related components such as path finding and agent related behaviour that are enabled (or included) in the definition of personality traits of in-game components.
- System 2 (*Acting Manager*) represents a method that receives messages from System III and distributes these messages to System I (*Speech Generator*, *Motion Generator* and *Appearance Generator*) for action to be taken.
- System 1 are the components represent the main action entities. These entities carry out the basic mechanics of the game. The *Speech Generator* provides the facility to play an appropriate sound file. *Motion Generator* processes the physics and animation data related to the in-game component modelled. *Appearance Generator* is responsible for rendering the graphics to the screen.

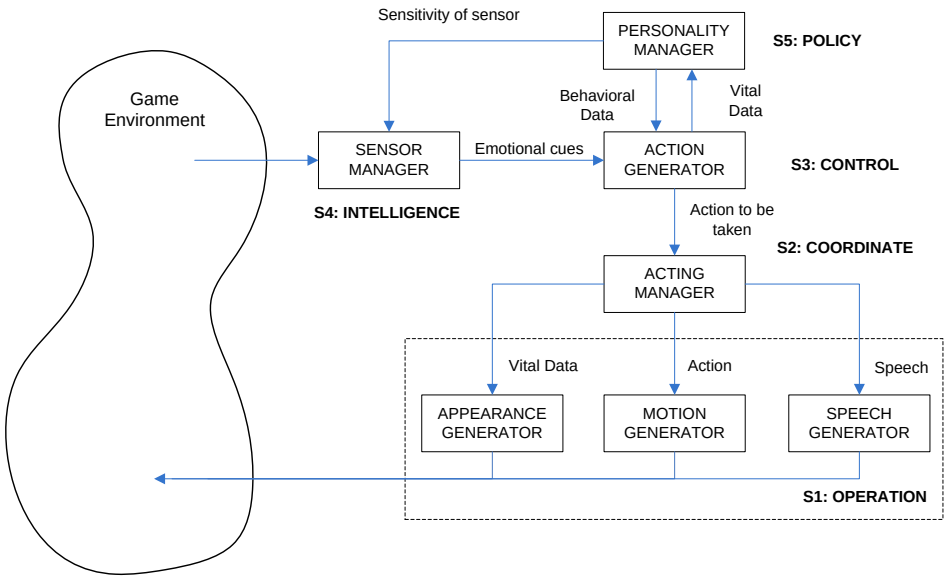

Figure 5.22: VSM as model for Object.

Our VSM description featured in Figure 5.22 is used as a high-level abstract model to represent game objects. Instead of specifying the software component, we will only use this model to identify the necessary data required to represent a game object and leave the implementation to the MDE developer. Based on the VSM model, we define a game object to consist of a set of *object attributes;* an *object appearance; object intelligence;* and it can perform a set of *object's actions.* These data are processed by the core game technology components defined in Section 5.4.2. The core game technology components are abstract form of Appearance Generator, Motion Generator, Speech Generator, Acting Manager, Action Manager, Sensor Manager and Personality Manager architected from a game technology perspective and not the game object's perspective. The game object data we defined earlier provide all the necessary information for the systems to operate. The operational and functional aspect of game object is dependent on facilities provided by the game software framework. Details of these data will be described further in the following sub-sections.

In computer games, game objects can be categorised into interactive game objects and non-interactive game objects based on their behaviours. Examples of interactive game objects include virtual things such as actors, items, consumables, enhancements and mechanical which the game player can interact with; while the non-interactive game objects include decorative, surface and structural which are virtual things use to define the boundary of the play space (see Table 5.1 for category description of Game Object). Each of these game objects has different configuration of the VSM model featured in Figure 5.22. For example game objects of *surface, decorative* and *structural* category would not require any data that would help describe game object's intelligence. Therefore game objects of those categories simply do not require data that describe intelligence. These classes of game objects are some examples of stock components defined to aid game designers to describe a game object and this list is expandable (see Figure 5.23). A full BNF representation of Game Object is available in Table A.4.

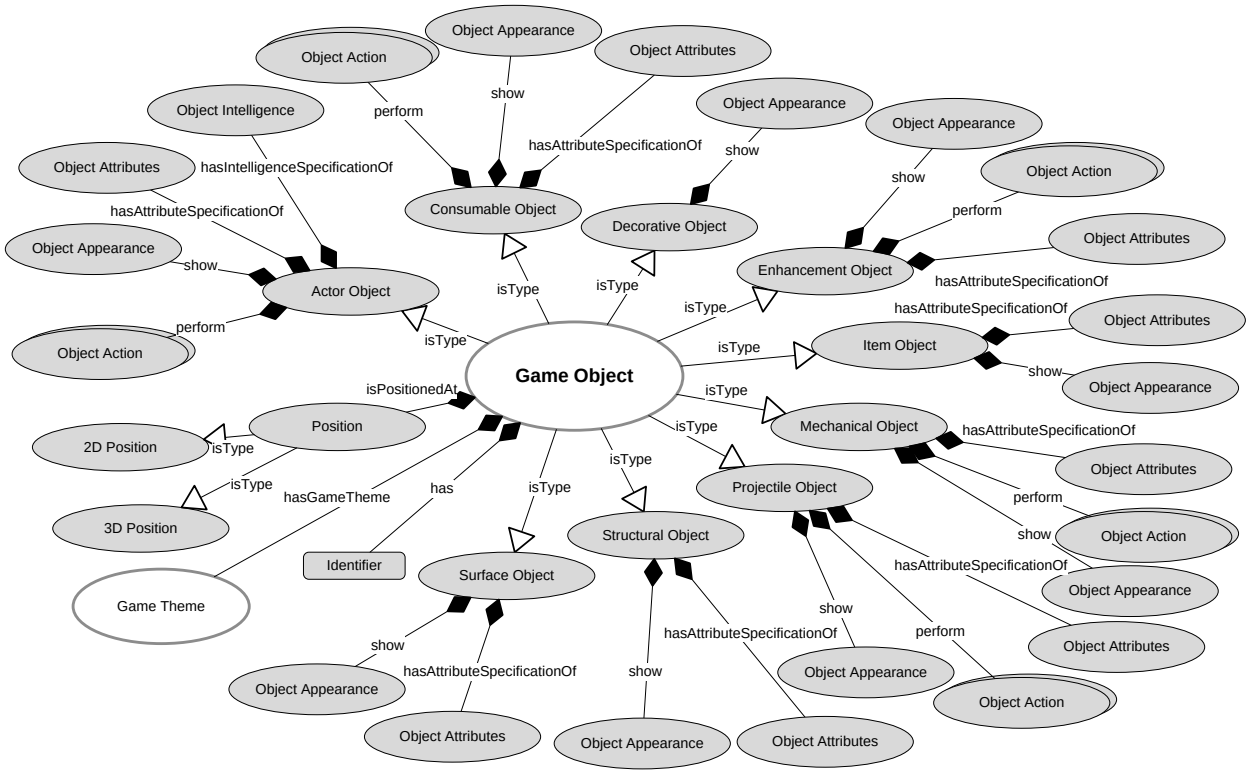

Figure 5.23: Ontology Diagram for Game Object.

Table 5.1: Types of Game Object

Category	Description
Actor	Actors are non-player characters (NPC) that populates the game-world. They can take any form of visual representation.
Enhancement	Enhancements are actor-dependent objects used to enhance the actor in terms of performance, vital or even visual appearance. Example of enhancements include knife, gun, torch light, clothing and shield.
Consumable	Consumables are also actor-dependent and some are meant to be used with enhancements such as first aid kit to increase the health of the actor. Examples of consumables include ammo, first aid kit, manna and grass as food for animals or harvested for hay production.
Item	Items are physically responsive objects which game-players can interact with. Some examples of item are glass window, chair and light bulb. Static items are regarded as decorative.
Mechanical	Mechanicals are game objects that consist of moving parts or switch controls which user can interact with such as doors, vehicle and switches.
Projectile	Projectiles are objects that propel with force on space. Example of projectile include bullet and soccer ball.
Structural	Structurals are any man-made structures that are used to fill up spaces in game-world. These structural objects can be buildings, bridges or a space within a building which are used as the set for defining scenario.
Decorative	Decoratives are static objects used to complete the sets. Examples of decorative include table, lights, photo frame and even in organic form such as trees and plants.
Surface	Plane is the surface for which all objects is attached to regardless of natural or man-made. Plane is regarded as roads or pavement on a city set, as floor in a Victorian house set, pond in the park or as grassland in traditional English farmland.

5.3.4.1 Object Attributes

Object attributes are data properties that represent a game object's attribute such as the vital statistics associated, physical state, cognitive state and ownership (see Figure 5.24). The object attributes

associated to a game object include a set of *vital signs*, a *position*, a *solidity state*, a *mass* and *inventory* (optional). Game designers can choose to represent the game object using a single vital sign definition such as life or further elaborate the vitality of a game object into interrelated attributes such as health, energy, strength, social and etc. The position of a game object is specified when setting up a game scenario and it changes depending on how game object reacts to the events in the game scenario and the dynamics in the game simulation. The solidity state defines if a game object is solid, whereas the mass affects the motion of the game object. The inventory is defined to hold a limited supply of consumable for a game object. For example, a rifle as an enhancement to an actor might be able holds 30 bullets at any one time. Similar concept can be applied to the actor whereby it can possess a number of weapons and cartridges of ammo.

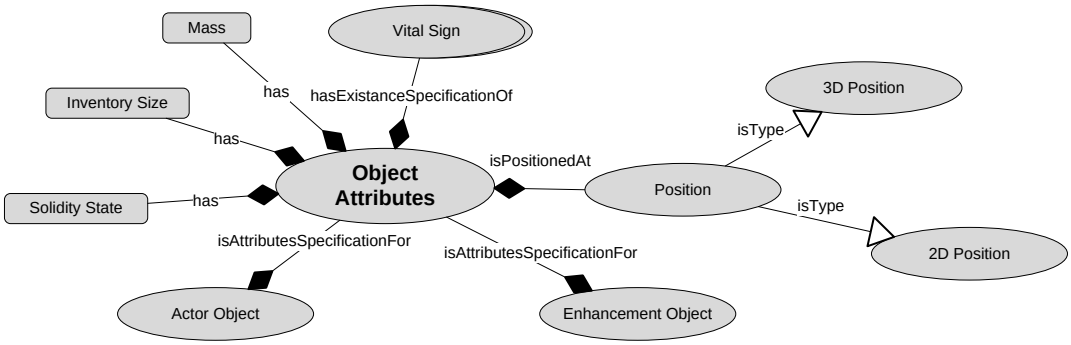

Figure 5.24: Ontology Diagram for Object Attributes.

5.3.4.2 Object Appearance

All game objects are represented visually with their own appearances. The appearance of a game object is composed of an *object image* and one or more *internal state projections* (optional). The object image is represented using one or more *image components* (see Figure 5.25). The image component represents one part of the object image and a collection of related image components complete the object image. Each part may have similar image components which the game player can use to personalise the game object and for game designer to create a variety of game objects to populate the game world. An *image style* is the combination image components that made up the object image and a game object can only be represented with one image style at any one time. The object image style can be represented using a 2D sprite or a 3D model with mesh and textures. Each object image style consists of a complete set of assets that project an appearance of the game object organised in an ordered path. The internal state projection displays specific object attributes using a front end display. In most computer games, display of internal state projection is limited to the use of simple

visualisation component such as text and bar which appears above the game object using a billboard. More complex forms of front end display can be used but should be used cautiously to ensure it does not add unnecessary processing load to the game software which in turn would affect the game-play and also the usability aspect of such a complex display have on the game.

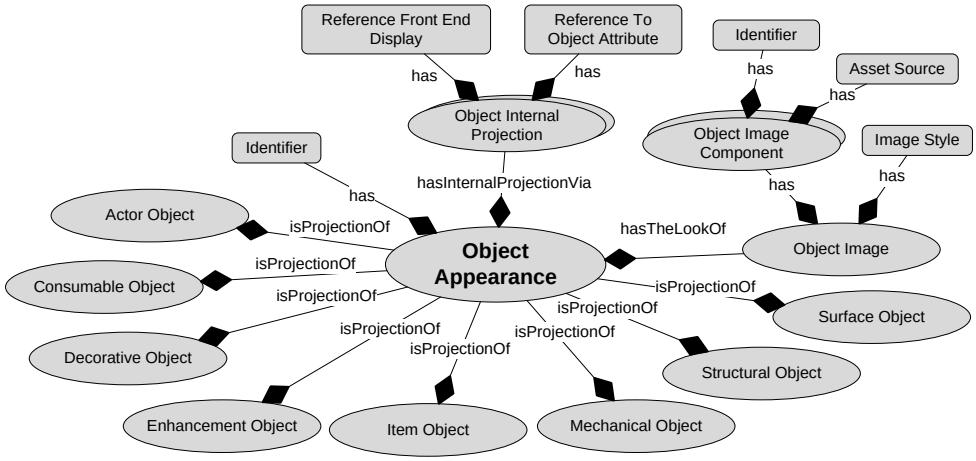

Figure 5.25: Ontology Diagram for Object Appearance.

5.3.4.3 Object Action

The action of a game object can consist of *motion*, *animation* and *sound*. In our Game Content Model, we regard motion and animation as two different concepts. We define motion as the actual translation of a game object's position in the game world, whereas animation is regarded as movement of objects moving parts in a position. Motion, animation, sound and vital update can be used on its own or can be combined to represent an action (see Figure 5.26). These components are described as follows:

- A motion is governed a force which moves a game object along a vector. It can be affected by the external forces which can increase or decrease the existing force the game object is subject to depending on the setting of the external forces. A force is diminished by the opposing force(s) or expires when the associated animation cycle completes.
- An animation defines the virtual performance of a game object without reference to the position of the game object as a whole. Definition of an animation sequence depends on the underlying technology supporting animation. For 2D sprite animation, an animation sequence can be defined by as the collection of 2D images. Whereas for 3D there are a set of techniques which developers can opt for frame-based and procedural. Other specific techniques for character animation are rag-doll physics, inverse and forward kinematics. The animation data for 3D are either supplied or generated programmatically.

- A sound is an aural form of action. It can be in the form of speech or non-speech. Each sound has an audio source and may be supplied with a text as the narration.
- A vital update revises the value of vital signs for a game object as a measure that takes account of the virtual effort of an action inflicted on the object or the virtual effort of performing an action. Virtual effort in this context refers to the decrement of values associated to vital signs defined by designers of the game.

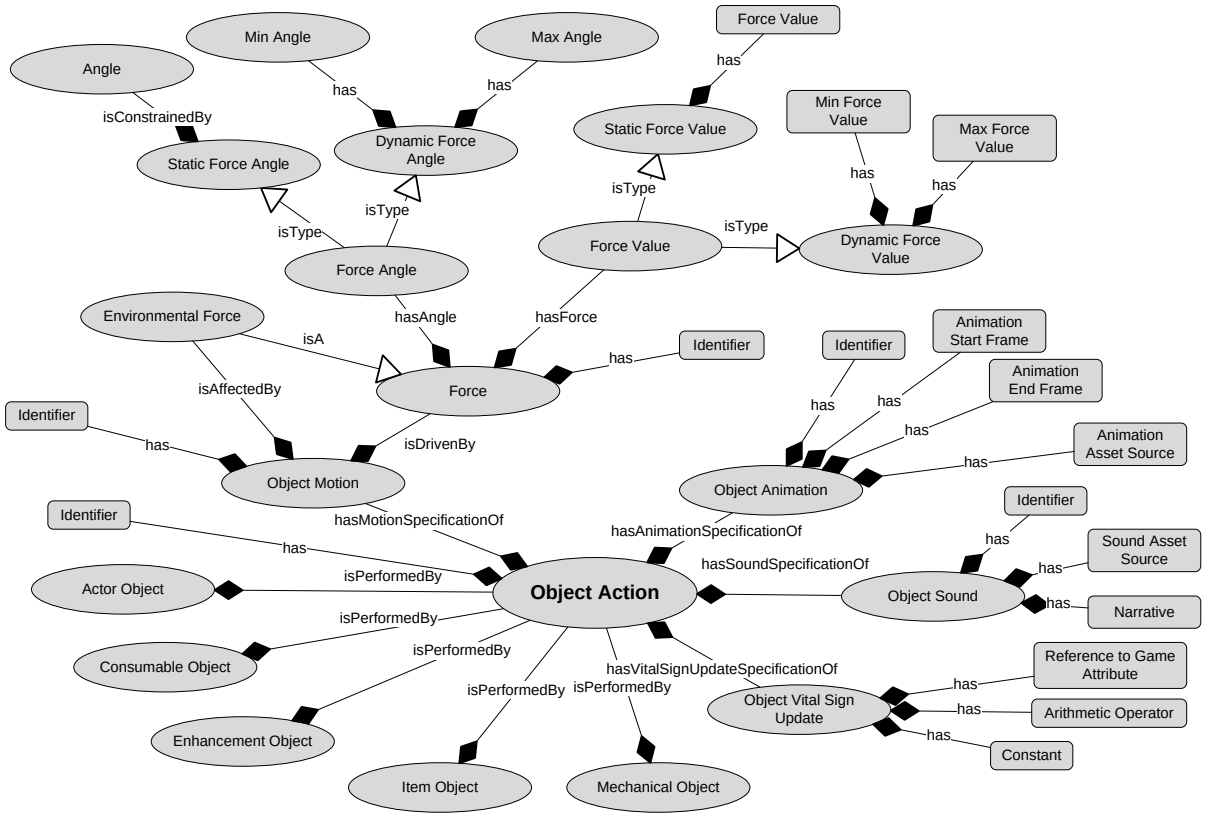

Figure 5.26: Ontology Diagram for Object Action.

5.3.4.4 Object Intelligence

The intelligence component provides a game object the ability to decide, navigate and even learn. One of the primary forms of intelligence required to be modelled by most game designer is decision making. A decision is composed of a *deciding condition* which is associated with an action. The deciding condition is closely linked to collision detection, direct input from the game player or events in the game scenario. These conditions are clearly labelled and are paired with an action of a game object to give the game object the ability to respond to events in a game scenario (see Figure 5.27).

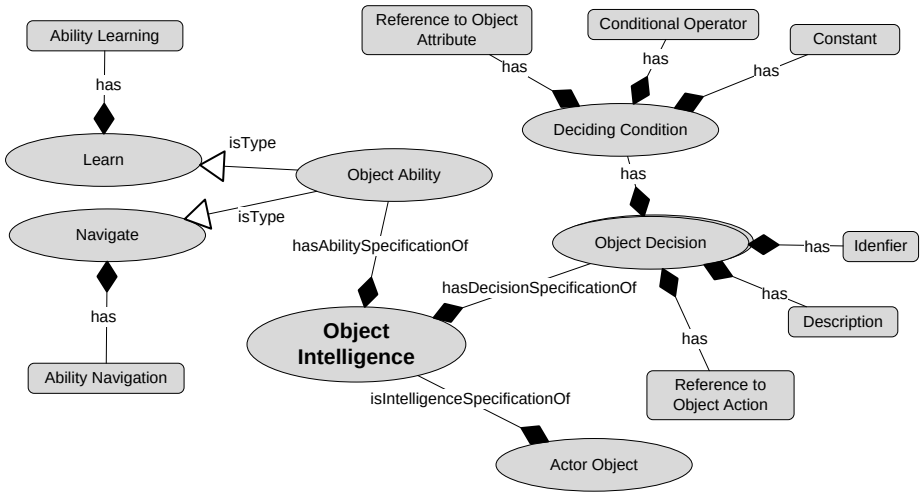

Figure 5.27: Ontology Diagram for Object Intelligence.

5.3.5 Game Scenario

The game scenario, as previously described in the game simulation (see Section 5.3.3), is a description of a situation which require game player to overcome a number of challenges in order to achieve the defined game objectives. It is also commonly termed as game level in gaming jargon. A game scenario, in our Game Content Model, is represented by a game environment, a set of game events (read more about game event from Section 5.3.6), a set of virtual cameras, a difficulty indicator and a set of game objectives (read more about game objective from Section 5.3.7) (see Figure 5.28). All interactions within the game scenario are regulated by the game rules (see Section 5.3.8) and the intelligence component of the game object (see Section 5.3.4). A full BNF representation of Game Scenario is available in Table A.5.

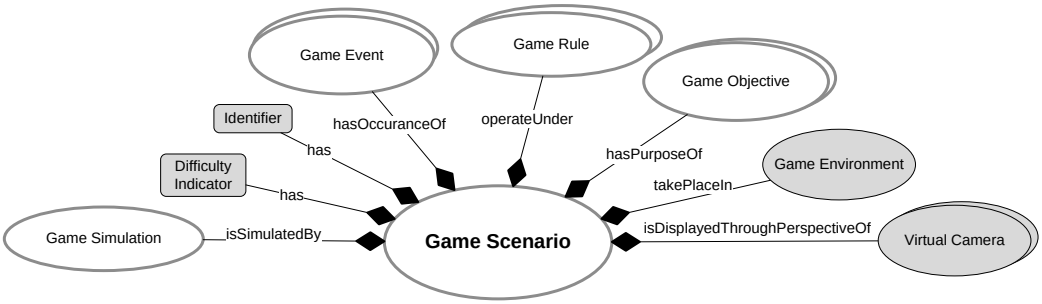

Figure 5.28: Ontology Diagram for Game Scenario.

5.3.5.1 Game Environment

The game environment represents the composition of the virtual world by populating the virtual space with *game objects, checkpoints*, *proximity triggers* and *lights*. (see Figure 5.29). Game objects are virtual

things that are carefully placed to create a spatial and social setting that form the game environment (see Section 5.3.4 for more details on game objects). An example of a game environment for Killzone 3 (Guerrilla Games, 2011) is shown in Figure 5.30. Checkpoints are also placed strategically within the game environment, but only used to mark positions in the virtual space which will be used in definitions of NPCs act in a game event. Checkpoints are only positional information and will not be shown visually. Proximity triggers, described in Section 5.3.1.3, are placed within the game environment but are not shown visually on screen. These are used only to evoke game events or to signify the achievement of a game objective. Lights are used to create "mood" to the game setting. The most usual form of lights are point light which illuminates all objects within the range in all directions, directional light which only illuminates objects within the range in a given direction, spot light which illuminates objects within the cone defined by the spot angle, range and angle and area light which illuminates objects within a specified area. In a basic lighting model, a light would consist of position, colour and intensity. Shadow is cast on objects within the range of the light. However, it is a very costly feature and developers often fake such an effect by "baking" the shadows on to texture to save computing resources. In our ontology, we omit shadow in our definition of light.

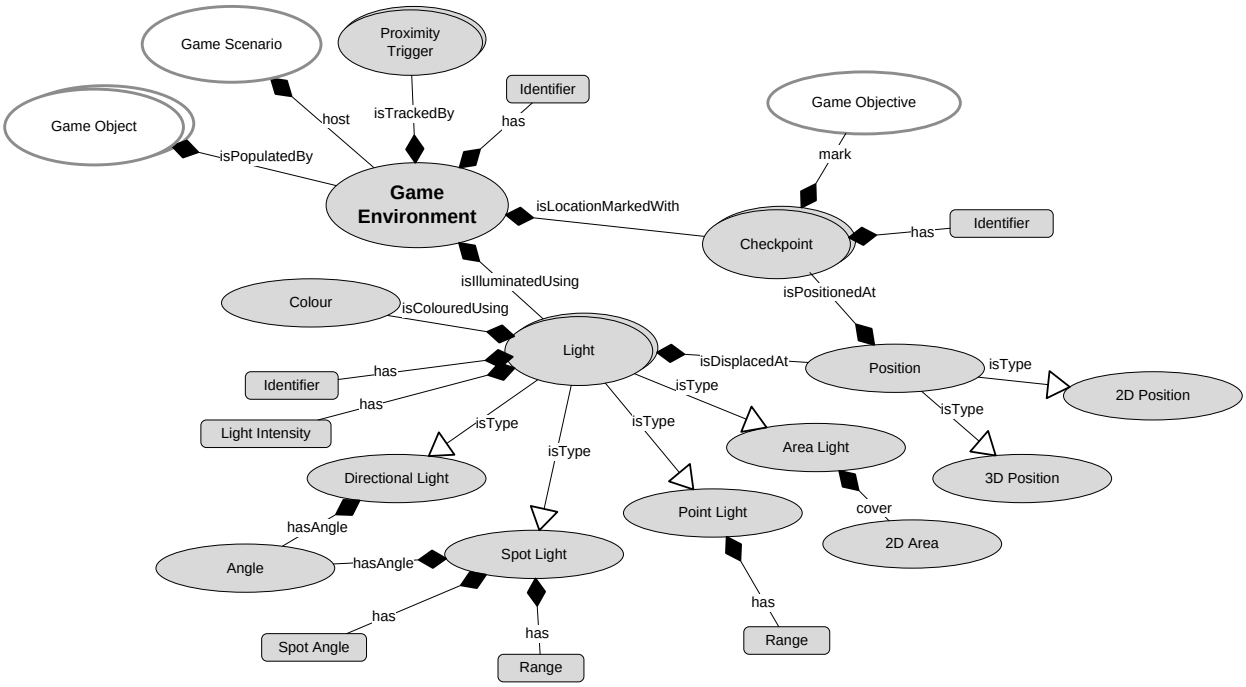

Figure 5.29: Ontology Diagram for Game Environment.

Figure 5.30: Game environment in Killzone 3. (Image obtained from http://4.bp.blogspot.com/-MgWqmqfqczE/Ta63EANhI9I/AAAAAAAAAQI/QpsLD-jI6Ts/s1600/Bilgarsk_Boulevard_W.png).

5.3.5.2 Virtual Camera

A virtual camera is the viewport onto the game world. In the context of game design, the only aspect that matters is the position of the virtual camera in the game world. A virtual camera can be mounted to an avatar to provide a first person view or hovering above the player's avatar to show a third person view. It can also be placed in a fixed location or animated along a path to provide a cinematic display. A game scenario can have more than one virtual camera to provide different viewpoints to the game environment, but only one virtual camera can be activated at once (unless is it part of the technological feature of the game to render multiple viewports on one single display which is seen in most multiplayer games on played on game console) (see Figure 5.31).

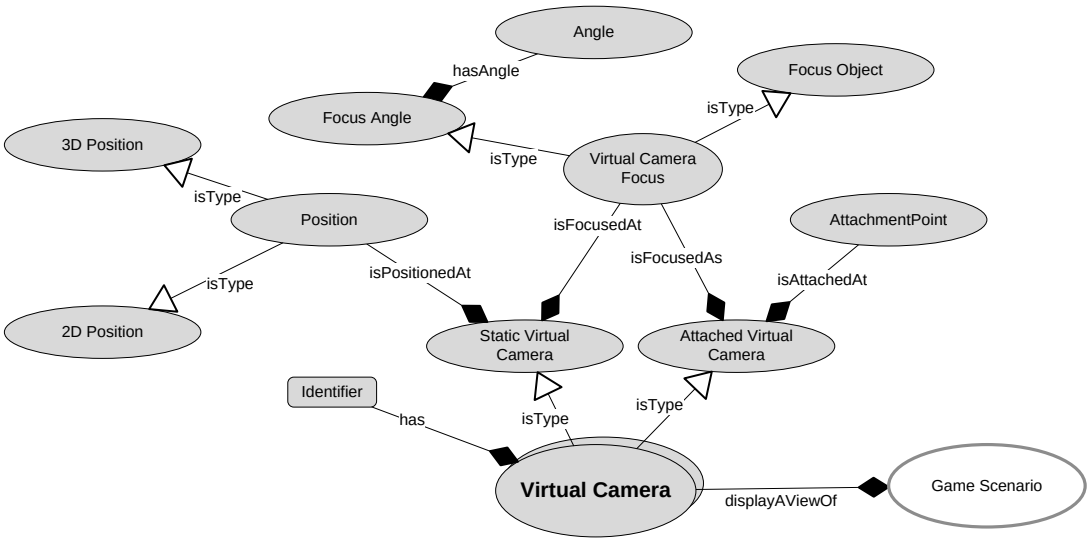

Figure 5.31: Ontology Diagram for Virtual Camera.

5.3.5.3 Difficulty Indicator

The difficulty indicator is a value associated to the difficulty of the game scenario. This value is set by the game designer and it is a mechanism to ensure that game scenarios within the game structure are organised in an increasing difficulty fashion. In addition, it can be used as a determinant for tuning the difficulty of a game scenario. In such a case, the difficulty indicator can be used as a multiplier that affect parameters such as game object generation frequency, number of game objects in the game scenario and the duration of game-play to adjust difficulty of game as it progresses.

5.3.6 Game Event

Game event is a happening associated with a game scenario. A game event is composed of a set of game acts and an event trigger which initiates the game event. Time trigger, proximity trigger and game mechanic trigger are the suitable types of event trigger for use when defining game events (read more about event trigger in Section 5.3.1.3). This is a more accurate approach to represent flow of events compared to Taylor, et al (2006) proposal. Our definition of game event permits designer the flexibility to define the number of times a game event happens once when game player repeatedly visit the same location. Each event is regarded as a state and the event are transitional conditions that govern the progression of the 'story' in the scenario. In order to monitor the game player's progress, we introduce the concept of *checkpoint* which checks against the conditions that governs the state of a game objective. There can be more than one game objectives associated with a game scenario.

Game act, which we adapted the concept from the performing arts domain, refers to the "acting" of a game object in a game event. A game act consists of a game object and a game acting script that describe how the game object should move, animate, sound and interact with other game objects in a game scenario.

A game acting script is composed of one or more acting coordinations that instruct how a specific game object acts in a defined sequence. For example, a game object can perform coordination-1, followed by coordination-2 and ends with coordination-3. The same script can also be assigned to other game objects that perform the same act.

The actual detail of an act is defined as acting coordination. The coordination of a game object can involve appearing (for generation and re-spawning of game object), animating an action, playing a sound, moving towards a checkpoint, interacting with another game object or a composition of these. Each of the coordinations in a script is executed one after another in the prescribed order. The

collection of game events defined in a game scenario will provide the flow to the game-play (see Figure 5.32). A full BNF representation of Game Event is available in Table A.6.

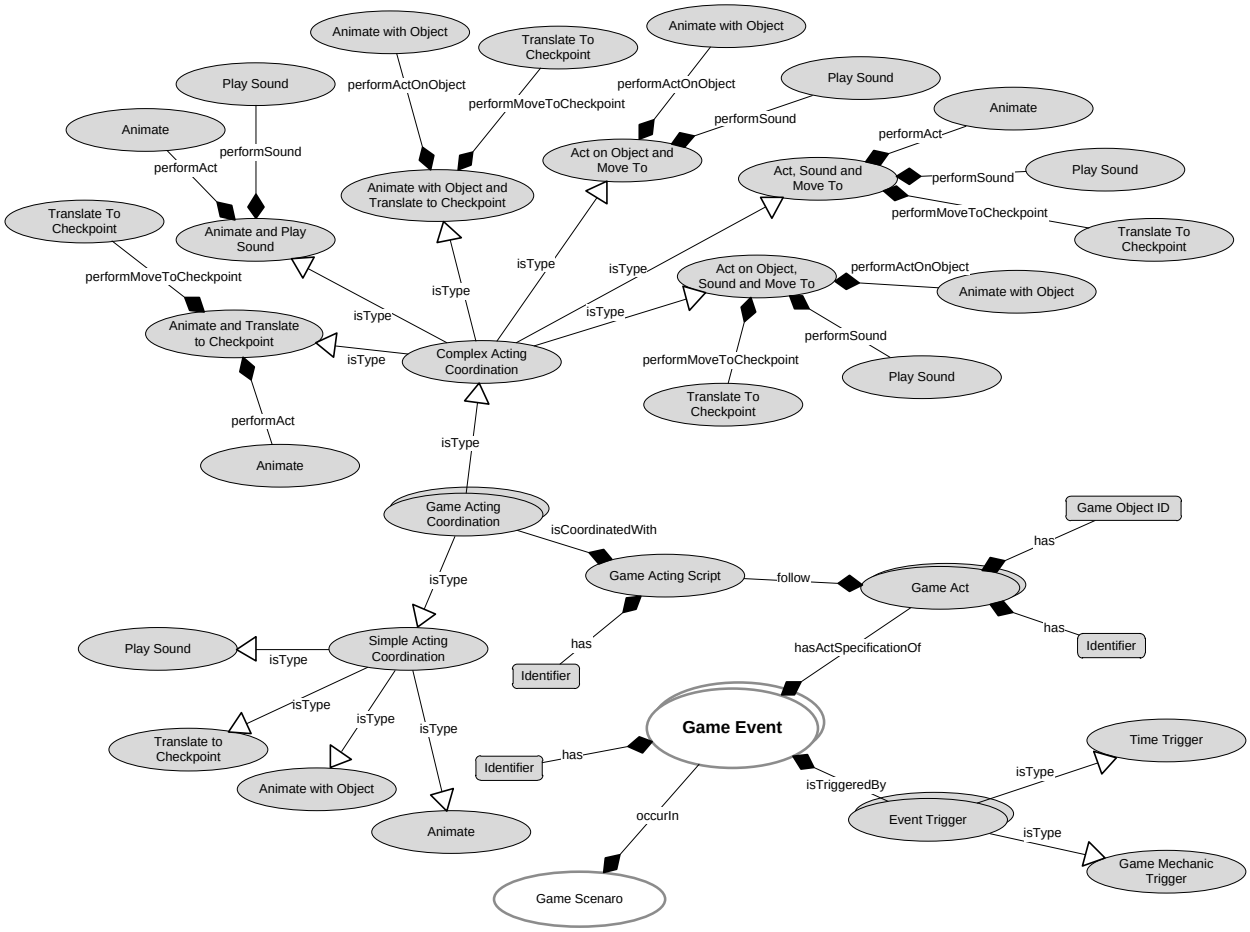

Figure 5.32: Ontology Diagram for Game Event.

5.3.7 Game Objective

A game objective is the goal associated with a game scenario. It is one of the components that define the game-play. The other components that define game-play are game-rules, game player and the game scenario itself. A game objective is represented using a *goal condition*. The goal condition checks a track-able value against a constant value defined by the game designer to determine if the game objective has been met or not. The game objective is marked as achieved once the goal condition has been satisfied. The game player wins the challenges presented in the game scenario if he/she manages to achieve all the game objectives (see Figure 5.33). A full BNF representation of Game Objective is available in Table A.7.

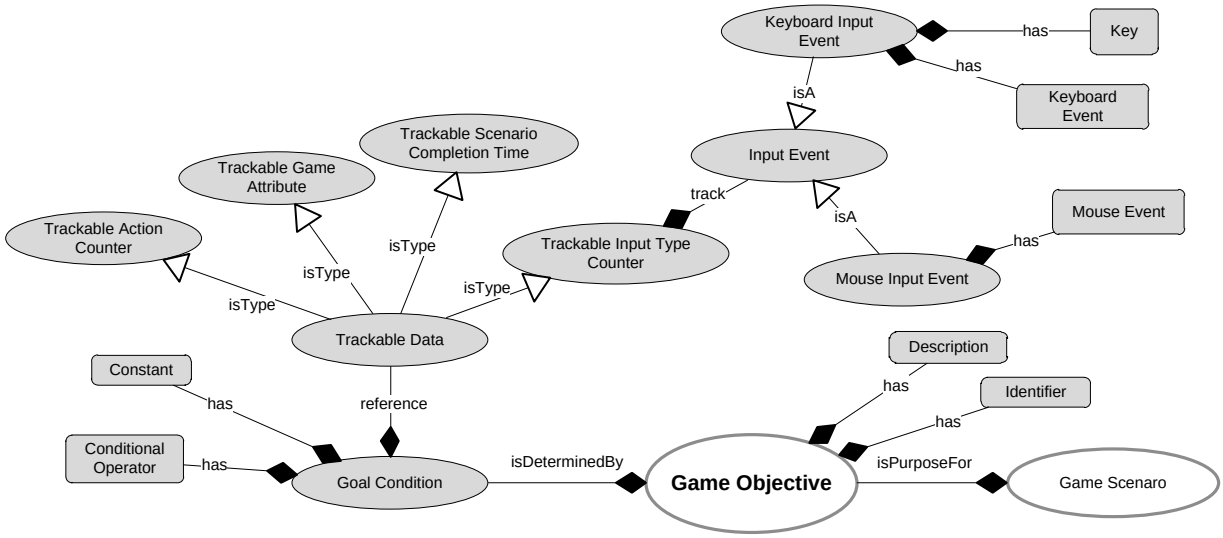

Figure 5.33: Ontology Diagram for Game Objective.

5.3.8 Game Rule

A game rule states the relationship between game objects and game world, and the effect of an interaction. In our model, a game rule can either be a *Game Scoring Rule* or a *Game Interaction Rule* (see Figure 5.34). A full BNF representation of Game Rule is available in Table A.8.

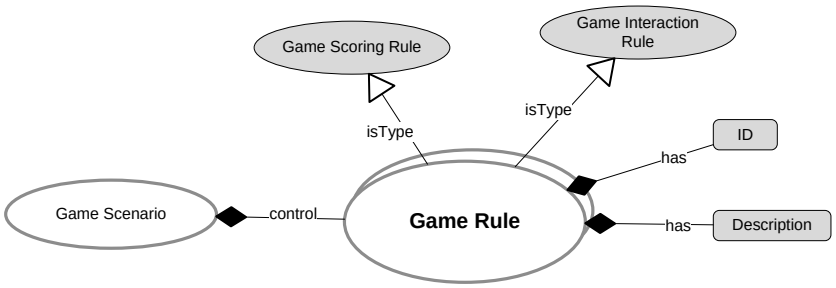

Figure 5.34: Ontology Diagram for Game Rule.

5.3.8.1 Game Interaction Rule

A game interaction rule differs slightly from a scoring rule. It dictates the *outcome* of the interaction from two game objects. Each game interaction rule has an *actor* which can be a game object, a class of game object or a group of game objects from different classes, a *subject* which represents a game object, a class of game object, a group of game objects from different classes or the game world and an *interaction condition* which refer to the state of actor or the state of the game world.

Each game interaction rule is paired with an *interaction outcome* which has a *matter* and an operation to add or subtract a value from the matter (see Figure 5.35). Matter in the context of game

refers to an *item* (e.g. key to dungeon door), an *attribute* (e.g. money) or permission (e.g. competition mode unlocked). Rewarding the player or any game object can mean giving (adding) or taking away (subtracting) an item or increasing (adding) or decreasing (subtracting) the value of an attribute. Conversely, punishment would mean consequences that would add more challenges to player or a game object.

An example for game interaction rule is ammo (actor) inflicts 10 points damage (interaction outcome) to Spartans and Elites (subject) whenever it comes in contact.

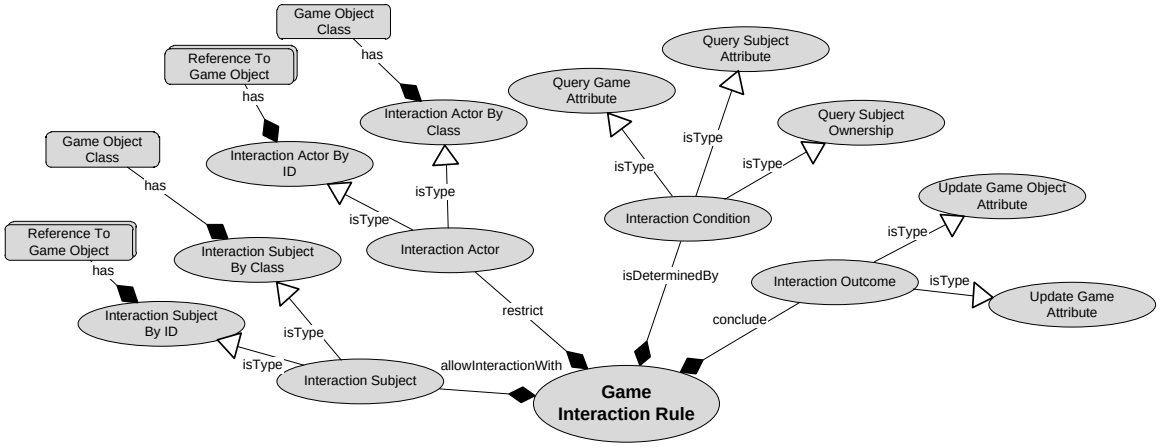

Figure 5.35: Ontology Diagram for Game Interaction Rule.

5.3.8.2 Game Scoring Rule

A game scoring rule only applies to game players and it defines what to be awarded to the game player when a *scoring condition* is met. Every game scoring rule has a scoring condition which has no direct relation with an interaction. It can be derived from the *state of a game object* (e.g. enemy is dead or fire is extinguished), the *input statistics* (e.g. accuracy is above 80%), the *time* (e.g. response is less than 5 seconds or level completed in less than 5 minutes), or the game objective (e.g. objectives 1, 2 and 5 are met) (see Figure 5.36).

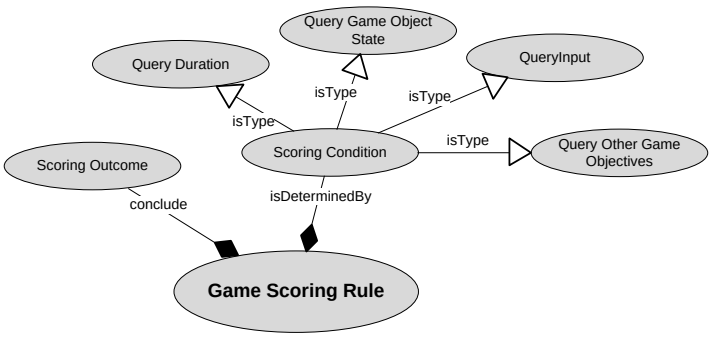

Figure 5.36: Ontology Diagram for Game Scoring Rule.

Figure 5.37: Halo Reach's Stockpile mode demands game player to bring flag into the team territory and defend it until timer reaches zero (Screenshot: http://www.bungie.net/projects/reach/images.aspx?c=59&i=25754).

An example of a game scoring rule in the Stockpile game mode for Halo: Reach (Bungie, 2010) (see Figure 5.37) is when a flag is brought into team territory and protected until the timer reaches zero (scoring condition) and the team is awarded with one flag point (scoring outcome).

5.3.9 Game Player

Game player is the user of the game application who provides inputs to the game system as part of the gaming activity. A game player is represented as an entity with an *avatar*, *game attributes*, an *inventory*, *game control* and *game records* (see Figure 5.38). A full BNF representation of Game Player is available in Table A.9.

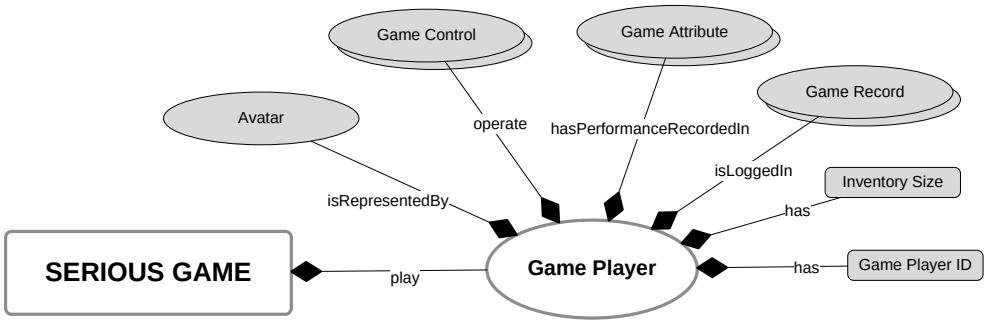

Figure 5.38: Ontology Diagram for Game Player.

5.3.9.1 Avatar

An avatar is a game object that represents the game player in the game world. It has direct relationship to the game control as input events of the game control interface are mapped onto the actions of the avatar. A game player can be assigned a fixed avatar or can choose to control a game object whenever

it is selected in the game world (dynamic). A game player is often assigned one fixed avatar throughout the game-play session in action, adventure, artificial life and vehicle simulation games (game player may have the option to select other character in game settings), whereas a game player can freely choose an avatar or multiple avatars to control in construction and management, strategy and sports games (see Figure 5.39). In most strategy games such as StarCraft II: Wings of Liberty (Blizzard Entertainment, 2010) (see Figure 5.40), the game player can group game objects into a single unit which he/she can command.

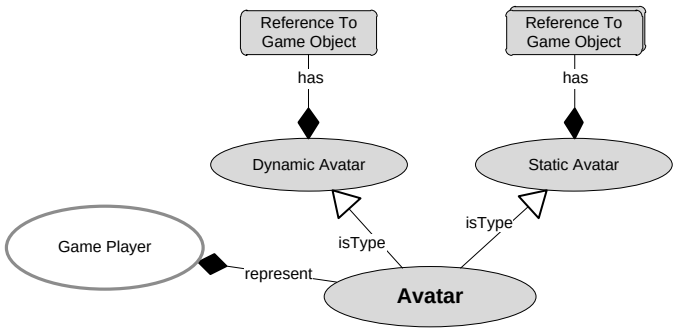

Figure 5.39: Ontology Diagram for Avatar.

Figure 5.40: Game player commanding 6 of the Reapers from Terran unit in StarCraft II. (Screenshot: http://www.sc2win.com/wp-content/uploads/2010/02/gameplay.jpg).

5.3.9.2 Inventory

The inventory presents the idea where the game player can have virtual ownership a number of game objects in the game as part of the game-play. An inventory can be a collection of weapons in a shooter

game or a collection of magic spells and potions in a role-playing game. In Resident Evil 5 (Capcom, 2009) (see Figure 5.41), the game player can hold 9 items in the inventory at any one time.

Figure 5.41: Inventory menu in Resident Evil 5. (Screenshot: http://www.ps3home.co.uk/userfiles/residenevil5-inventory-menu.jpg).

5.3.9.3 Game Attribute

Game attributes are data representing the existence and achievement of a player. These data can be either a vital sign or a score. A vital statistic is the value associated to the chance that a player before he/she loses the game, whereas a score is a record of performance which can be derived from time spent or remaining or the number of game objects interacted with either indirectly or directly via an input event. For example in Pacman game (Namco, 1980), the game player has three lives or chances (vital signs) to play the game and his/her achievement is recorded as score (refer to Figure 5.43). For most modern games, the game player's vital signs are often linked to the avatar. Take StarCraft II: Wings of Liberty (Blizzard Entertainment, 2010) (see Figure 5.44) as an example. Here the game player's vital sign is represented by the entire military unit which includes resources available, whereas the score gained can be calculated from resources, units and structures. Game attributes can be simple or modelled in a complex way to demonstrate dependability on one another depending on the needs of the game designer (see Figure 5.42).

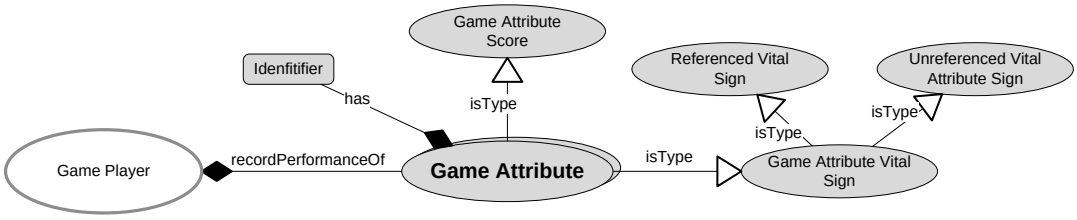

Figure 5.42: Ontology Diagram for Game Attribute.

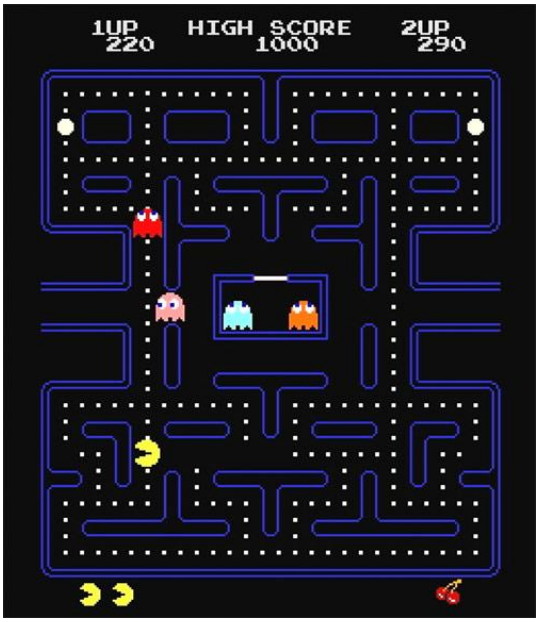

Figure 5.43: Classic Pacman game requires game player to collect all pellet and power-pellet in each game scenario. (Screenshot: Midway Games)

Figure 5.44: StarCraft II records resources gathered, units raised and structures built. (Screenshot: http://farm3.static.flickr.com/2709/4366787241_2e59df7ded_z.jpg).

5.3.9.4 Game Control

The game control provides the game player the medium for controlling game objects. Game player's input is captured via *game control interface* which can be either a *hardware interface* or a *GUI* (read more about GUI from Section 5.3.2.2). Examples of hardware interfaces include *keyboard*, *mouse*, *gamepad*, *joystick*, *motion sensor*, *camera* and *microphone*. There are also specialised game hardware interfaces available in the market such as the dance mat from the Dance Dance Revolution game (Konami, 1998) and the guitar controller from Guitar Hero (Harmonix, 2005), but underlying such innovative product designs is an altered form of a gamepad that provide players alternative way to

input and better game immersion. Hardware interfaces and GUIs have their respective input events. For example, a mouse has input events such as move, roll over, roll out, left button up, left button down, right button up and right button down. Each input event can link to one or more actions depending on the state of the game object. For example, the button 'x' triggers 'action 1' when game object is in 'state 1', it triggers 'action 2' when game object is in 'state 2'. Such complex input configurations are often seen in console games due to the limited input available on a gamepad (see Figure 5.45).

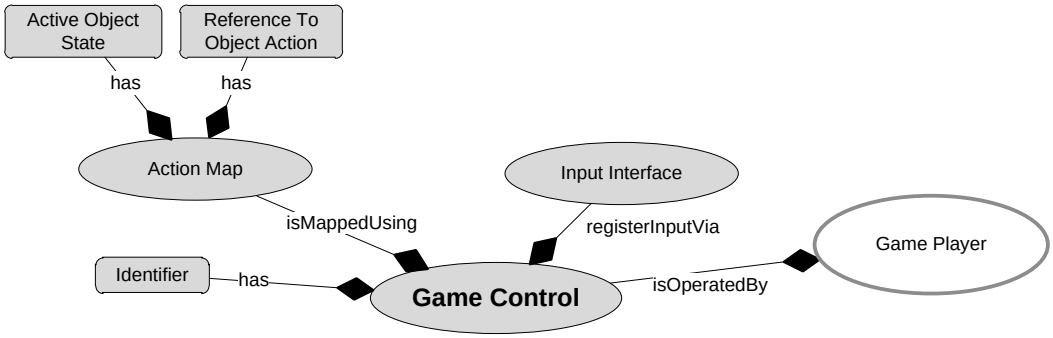

Figure 5.45: Ontology Diagram for Game Control.

5.3.9.5 Game Records

Game records are a log of the game player's achievements in the game. Although it is an understated aspect of game design, most modern games do record game player's achievement and progress in the game. Each game record is associated with the *game results* of a game scenario. A game result can either be in the form of *raw results* or *computed result*. The raw results are obtained directly by summing up traceable data such as input event, game attributes from game player and time whereas *computed result* whereas traceable data is subjected to computation. An example of raw result in StarCraft II game is the resources gathered by the game player during the game session, whereas the computed result can be average unused resources (see Figure 5.46).

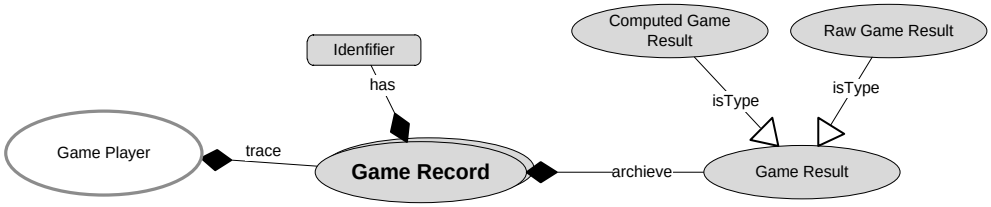

Figure 5.46: Ontology Diagram for Game Record.

5.3.10 Game Theme

Game theme describes most of the art requirements related the game through expressive written text. It is specified as part of the definition of a game object. The art requirements include visual, sound and even style of the narration that defines the "look and feel" of the game object. These are media content on its own which are produced externally by professionals and conform to the format of the game technology. A BNF representation of Game Theme is available in Table A.10.

5.4 Game Technology Model (GTM)

In the past, games were written as singular entities in assembly languages that were tightly coupled with the underlying hardware platform to utilise hardware resources efficiently and hence provide a seamless gaming experience (Bishop, Eberly, Whitted, Finch, & Shantz, 1998). However, this approach permits little code reuse and low code scalability which results in complex games being both costly and timely to develop. Coding these games is also a highly specialised skill.

A game engine or game software framework is the technical and economic solution for writing modern, complex games. It consists of subsystems or software components that perform a number of distinct tasks (such as graphic rendering (2D or 3D); game physics computation such as collision detection; collision reaction and locomotion; programmed intelligence; user input; game data management and other supporting technologies) to operate the game software. These software components are built to manage, accept, compute and communicate data with other software components without fail. Not all game engines support the entire feature set required for all game genres, since integrating these technological components under a single framework would be a prohibitive task. Furthermore, not all computer games software will require the entire collection of software components to function.

The early streamlined approach to game engine architecture (illustrated in Figure 5.47) was composed of the software components that handle input, audio, graphic representation and the dynamics (game mechanics) (Bishop, et al., 1998).

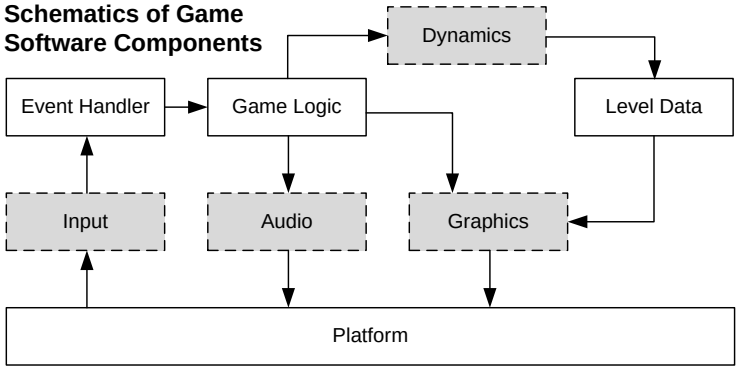

Figure 5.47: Aspects of game software illustrated in shaded rectangles are elements of a game engine while game logic and level data are regarded as *content* that defines a game (Bishop, et al., 1998).

Modern day game engines are capable of computing complex 3D scenes with dynamic objects, rendering realistic graphic, planning and deciding actions for non-player characters (NPC), and supporting multiple players concurrently over a network. The architecture of the Delta3D engine documented by Draken, McDowell & Johnson (2005) in Figure 5.48 exhibits the addition of software components such as character animation, scene graph and networking to assist the development of modern 3D games with multi-player support.

Current generation commercial game engines such as Unreal Engine, CryEngine and EgoEngine are packed with more advanced features and sophisticated tools that enable creation of high quality game software. The game engine architecture explained by Gregory (2009) (see Figure 5.49) illustrates the typical logical architecture of a modern 3D game engine. It is designed for maximum reusability (software components are shown shaded) and interoperability across different platforms (through the Platform Independence Layer).

The Game-Specific Subsystems and Gameplay Foundations are components that can be re-written to adapt other components for other game genres with minimal changes to the remaining software components (some changes are still necessary as these components may be written specifically for the chosen game genre for optimal performance and reliability).

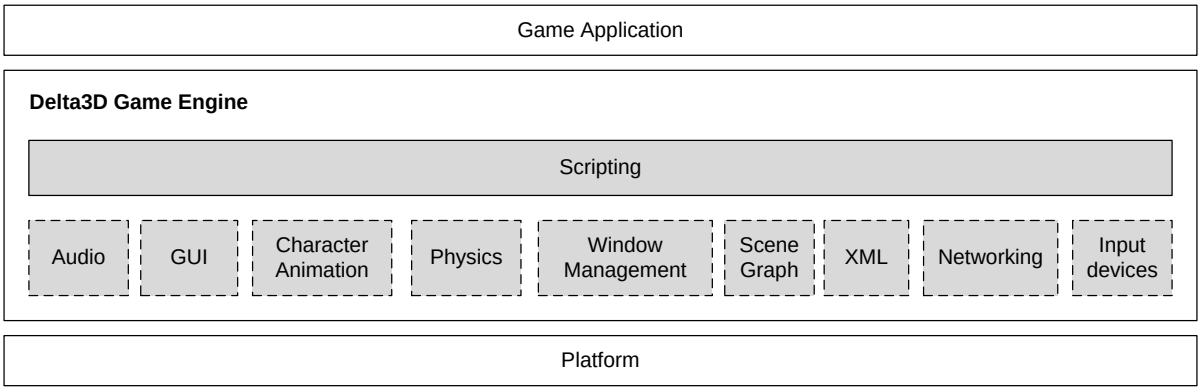

Figure 5.48: Architecture of the Delta3D Game Engine (Darken, et al., 2005).

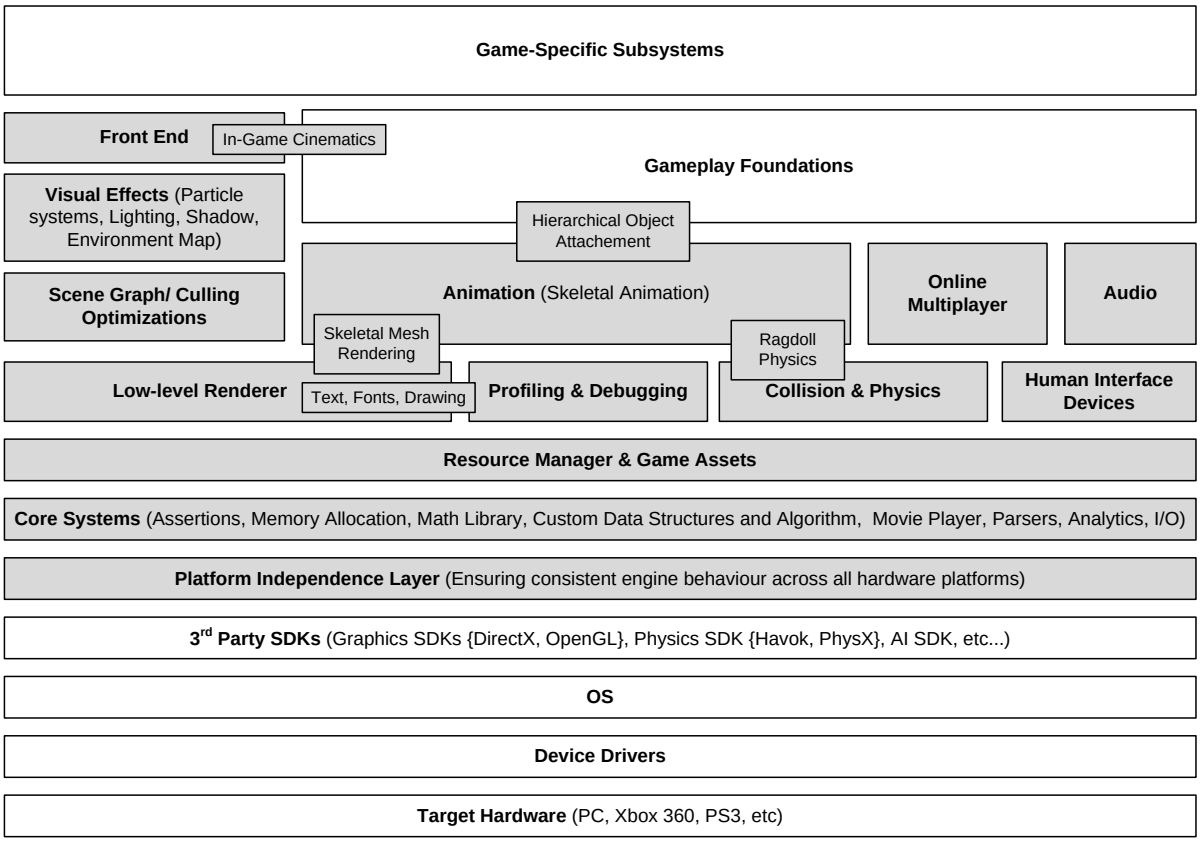

Figure 5.49: Overview of Commercial Grade Game Engine Architecture (Gregory, 2009).

The responsibility of the Platform Independence Layer is to ensure that all components in the game engine behave consistently across different hardware platforms. This is similar to the functionality of a Platform Independent Model from a model viewpoint.

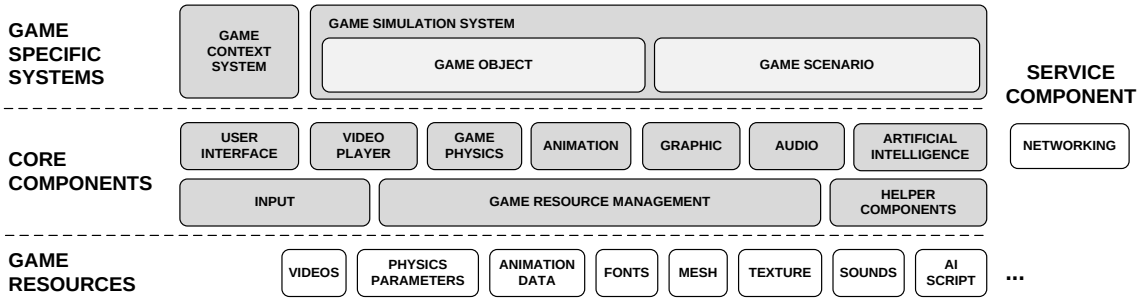

Figure 5.50: Overview of Game Technology Model.

The Game Technology Model in our model-driven framework represents games in a manner that is computationally independent of any operating platform. This is responsible for ensuring all components in the game engine behave consistently across different hardware platforms. Our proposed Game Technology Model in Figure 5.50 features two primary layers: the *Game Specific Systems* layer and the *Core Components* layer. In the following subsections, we will describe the systems and components in each layer and represent the computer game software in high-level formal notation to ensure the Game Technology Model is not tied to any specific technique or platform. Platform specific details will be added to the model in when Game Technology Model is translated into Game Software Model. Details implementation of software will be generated by generator from predefined code templates through mapping techniques.

5.4.1 Game Specific Systems

The Game Specific Systems layer consists primarily of the *Game Context System,* which sets up the game and manages dynamic switching between the presentation context and simulation context, and the *Game Simulation System,* which populates the world with game objects (both static and dynamic) and triggers the game events that form part of the game-play within a particular game scenario. It uses facilities provided by the core components layer (see Section 5.4.2 for more details) to enable smooth running of both the Game Context System and Game Simulation System.

5.4.1.1 Game Context System

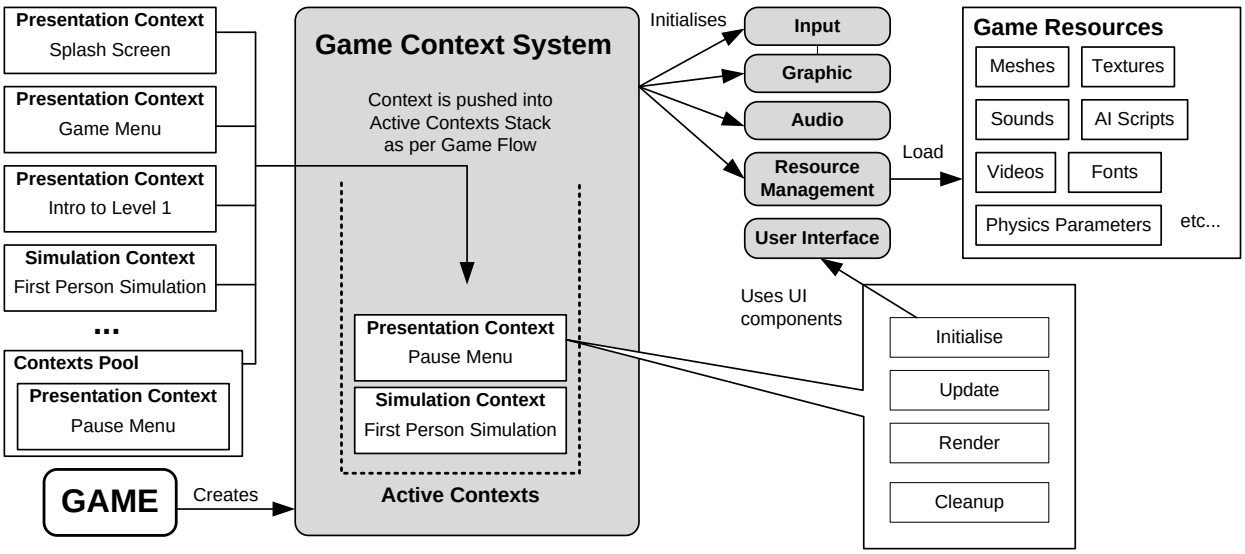

Figure 5.51: Game Context System

Looking deeper into our Game Technology Model (see Figure 5.50), the responsibilities of the Game Context System are to set up the application by initialising the input hardware, graphic hardware and audio hardware, specifying access paths for resources, switching between different contexts and freeing up hardware resources prior to shutting down the game application. In our Game Content Model, we break down a game into sections known as *game context*. A game context describes the type of game content to be presented to game players which can be either in the form of a *game presentation* or a *game simulation* (see Section 5.3.1.1). As described in Section 5.3.5, game scenarios are contents to a game simulation. Separating game scenario from game simulation enables us to use the same game scenario in a different yet compatible game simulation setup. These contexts are loaded into the *Game Context System's Active Contexts Stack* based on the flow of the game defined. This stack-based approach allows multiple contexts to be rendered in the correct order, producing a layered effect as described in Carter, Rhalibi , Merabti & Price (2009).

Each context has its own methods for: initialising the data or components required; an update method which updates state of the context; and a render method which presents the visuals on the screen. Update and render routines are invoked at 30-60 time steps per second. The updates can also be multithreaded using fork-join parallelism (Kim & Agrawala, 1989). The clean-up method is invoked when a context is popped out from the active contexts stack to free off resources. Game contexts which are frequently used can be placed in a resource pool to avoid constant loading and unloading which could result to performance slowdown. UML diagram Game Context System (Game

Context Manager) and Game Simulation System (Game Simulation Manager) is available in Figure B.1 (Appendix B).

The transition from one context to another is triggered by the event trigger (refer to Figure B.2 in Appendix B). Event trigger monitors a defined event which can be an application event (GameMechanicsTrigger), an input event (InputTrigger) or a time-based event (TimeTrigger). When the condition of the event is met, it notifies the Context Manager to push out the expired context and pop in the next context into the Active Context stack. Each event trigger is synchronised with the main update method which can be monitored by a manager such as the Game Mechanic Trigger and the Input Trigger whereas Time Trigger and proximity trigger are monitored in the respective update methods (which are also synced with the main update method) of the class it composed of.

At the top-most level of the game software is the computer game application itself which uses the game context system and provides the necessary interfaces for the game player to interact with the game. The game player component, as described in the Game Content Model, is composed of a reference to a game object which acts as an avatar, a set of attributes that determines the game player's vitality or performance, a set of control maps which wires input events to the associated action of the avatar, a structure to store virtual items that player can own in the game and a statistical log of the player's performance which are integrated into the game (refer to UML diagram for Game Player in Figure B.3, Appendix B).

A game can be dependent on a fixed time step or variable time step. In our Game Technology Model, we favour for simplicity over complexity and hence have opted for fixed time step implementation. Within the game update method it checks for any input and application events and synchronises with the update method for each context.

```
Main()
{
      gameRunning = true
      fps = 30
      delay = 1000/fps
      gameTime = 0
      sleepDuration = 0
      nextTick = GetTickCount()

      While (GameRunning){
            Game.Update()
            Game.Render()

            nextTick =+ delay
            sleepDuration = nextTick - GetTickCount()

            If (SleepDuration >= 0)
            {
                  Sleep (sleepDuration)
            }
      }
}
```

Figure 5.52: Fixed time step game loop

5.4.1.2 Game Simulation System

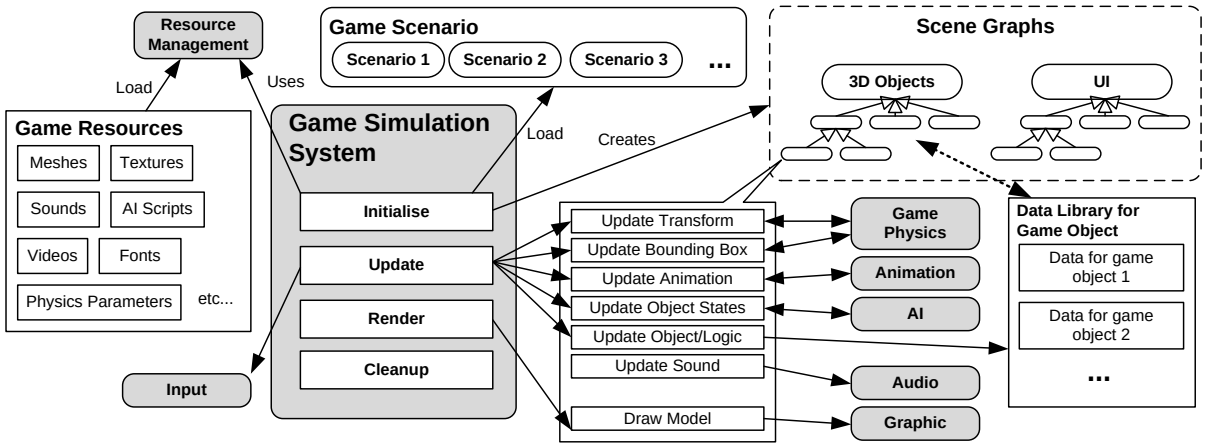

Figure 5.53: Game Simulation System

When an instance of a simulation context is made active, the Game Simulation System (see Figure 5.53) loads in the associated description of game scenario as part of the initialisation routine and plug-in the relevant components and its settings to operate the simulation which include environmental forces which affects the game physics. A game scenario, in our model, is represented by a game environment (that is composed of a collection of game objects), a set of game events, a set of virtual cameras, a difficulty indicator and a set of game rules (which dictates the outcome of the interaction from two game objects) (Tang & Hanneghan, 2011b). Game objects required for construction of the game environment are created and organised in a scene graph (see Figure B.4 for SceneGraph UML Diagram in Appendix B) allowing the rendering of graphic components in the correct order. Data

associated with game objects are stored in a collection which can be fetched and updated directly rather than having to traverse through the scene graph.

The Game Simulation System maintains two scene graphs: a scene graph for media, GUI and FED (2DGraph scene graph); and a scene graph for proximity triggers and game objects which made up the game environment (GameEnvironment scene graph is defined within the GameScenario class). The scene graph that stores game environment will be rendered to the image buffer before the 2D scene graph to allow final construction of render frame which has the media, GUI and FED overlaid on top of the game environment (see Figure B.5 in Appendix B for pseudo-code illustrating update and render method implementation in simulation context; and Figure B.6 shows the pseudo-code for recursively updating the nodes in scene graph. Similar algorithm can be used for rendering all nodes in a scene graph.).

At each update routine of the Game Simulation System, all game objects have their transform and animation updated. Object state, game data and other computationally demanding process such as collision check of the game objects are updated at different time intervals to avoid performance slowdown. Dynamic objects which no longer exist are removed from scene graph and data associated are destroyed during runtime. The events defined in the game scenario are also triggered in the update routine of the Game Simulation System. All update and render routines are in synchronization with the main game loop.

5.1.1.1.1 Game Scenario

A game scenario holds all the relevant data and data structures that represent a level in a game. Most of the description of a game scenario can be represented as data but descriptions such as game interaction rule and game scoring rule are automatically embedded in the respective game object *Update()* method whereas events are encoded in the game scenario *Update()* method for optimal performance using our MDE tool (see Figure B.7 in Appendix B for UML representation of Game Scenario). In addition to triggering game events, the *Update()* method in a game scenario checks if all game objectives are met and updates the game objects within the game environment scene graph. An example of Update() method for game scenario is available in Figure B.8, Appendix B.

5.1.1.1.2 Game Object

A game object is the primary data structure which will be processed in the Game Simulation System using the facilities provided by the game component. Game objects consist of attributes that hold

values of existence (vital signs) such as objects attributes, position, mass, solidity state and size of inventory (which defines the amount of objects associated with it), and behavioural characteristics which are represented in the form of motion, action, attribute updates, sound and intelligence. These can be represented by three distinct classes namely, Actor, DynamicObject and StaticObjects. In the software context, we can abstractly represent the nine categories of game objects described in Section 5.3.4 into Actor, Pickup and Static. Actor and Mechanical categories fit into the Actor Class. Consumable, Enhancement, Projectile, Item, Structural and Surface categories of game object are best represented using the DynamicObject class, while Decorative category of game object is represented using the StaticObject Class as illustrated in the UML Diagram for Game Object in Figure B.9, Appendix B.

In terms of mapping the game object from Game Content Model to Game Technology Model, the object attributes and animation sequences are easily mapped from the definition in Game Content Model into the structure. Action definitions of the game object such as animation, sound and motion can be invoked from the *Update()* method. These are grouped into the relevant action state of the game object as defined in the Game Content Model. This approach allows easy pairing with input event or decision making in AI. The input or AI components will only need to change the *activeState* of the object to trigger the paired action. Each action is marked in a conditional statement using a unique identifier as illustrated in Figure 5.54. At each game object update call, it checks the *activeState* which is changed when certain input event is triggered or AI decisions are called. Input events are updated at the main game loop whereas AI routine is done right before the invocation of the actions. For a more detail example, see Figure B.10.

```
If(this.actionState == 2)//Pick
{
      collisionObject = PhysicManager.checkCollision(this, GameEnvironment)
      If(collisionObject.type == "HealthPack")
      {
            AnimationManager.SetSequence(this, "Pick")
            this.health.add(25.0f)//health is a vital
      }
      ElseIf(collisionObject.type == "Key")
      {
            AnimationManager.SetSequence(this, "Pick")
            this.inventory.add(object);
      }
}
```

Figure 5.54: Example of Action definition in a Game Object

Within each action, a further query can be invoked to determine the actual state of an object. This could be in the form of collision checking to determine if a game object is within the boundary of another game object (which is defined by a game interaction rule in Game Content Model) or checking an attribute's value or inventory of the game object as illustrated in the pseudo-code in Figure B.10. An action, as described in the Game Content Model, consists of method calls to the relevant components such as physics and animation to update and transform the game object which is automatically transformed from specification of game design in our framework. It also consists of commands to invoke playback of audio files and relevant data updates to vital and inventory. The main update call will traverse the game environment scene graph and update all the states, data and transforms before executing the render routine.

The *Render()* method of a game object processes and displays the object appearance on the screen using facilities in the renderer component. In conventional approaches, all transforms are computed prior to the actual rendering process. In our data-driven approach, all the transform and rendering is computed by the renderer through an overloaded *Render()* method. This approach separates game logic from component computation simplifying transformation and generation of code.

```
Function Render(Component RenderManager, VirtualCamera camera){
      If (camera.WithinFrustum(this))
      {
            RenderManager.Render(this, camera)
      }
}
```

Figure 5.55: Pseudo-code for Render method in a Game Object

5.4.2 Core Components

Components of game engines vary depending on technology features and are often constrained by particular game genres, technology platforms (hardware) and visual dimension of the game world (2D or 3D), but there are some common technology components across all game engines. These are the core technologies that enable the creation of a variety of game software solutions, each featuring creative and compelling content. The core technologies we identified are *graphics (renderer), animation, audio, input, game physics, user interfaces, networking* and *game resources management*. In the following subsections we describe these core technologies in brief. For the purpose of our Game Technology Model, we treat each of the components as a "black-box" for processing respective game data. In the following subsections, we describe the features and functionality of each core components

and define the necessary interfaces which encapsulate the detail implementation for processing the game data defined in Section 5.4.1.

5.4.2.1 Renderer

The renderer is key component to any 2D or 3D graphic engine that is responsible for graphics-related computations and rasterised screen output. The 2D renderer provides the interface for graphic hardware and draws 2D graphics, whilst the 3D renderer provides additional functionalities such as loading 3D models, rendering and managing textures, applying different type of materials and blends to the texture, rendering static and dynamic lighting in the scene, displaying viewports and virtual screens, and providing control to the virtual camera. Other features which are found in a renderer also include particle systems and post-processing used for visual effects. In recent years, 3D graphic engines have gained significant popularity over 2D graphic engines due to the consumer demand for 3D games, whilst most 3D engines also support creation of 2D games through some clever exploitation of the technology, such as orthographic projection (Gregory, 2009). However, 2D graphic engines are still relevant, particularly on lightweight platforms such as the mobile and web platforms.

In our Game Technology Model, the renderer is initialised, set-up and shut-down by the Game Context Manager. This configuration allows us to plug-in different types of renderers such as DirectX, OpenGL and ORGE in the software system. During initialisation renderer is paired with the graphic device and necessary setup information such as draw mode is defined. In our data-driven approach, the renderer takes in various data structures and renders it to the image buffer before displaying to screen instead of embedding the rendering instructions in each object's *Render()* method as illustrated in the UML diagram in Figure B.11. Each of the render-able objects defined in the Game Content Model share a common interface named *iRenderable* which consist of a render method as shown in Figure B.4. Within each of the render method data structure of the object is passed in as parameter to the renderer for processing. Platform specific details required by the renderer will be filled in when Game Technology Model is translated into Game Software Model. Details implementation of each render-able object will be generated by generator from predefined code templates through mapping techniques.

5.4.2.2 Animation

The animation component is responsible for determining the next pose of a 3D model or the next frame of a sprite, independent of its spatial position within the world. A 2D animation component

would extend the functionality of a 2D graphic engine to include management of animation state which is why many would regard the animation subsystem to be part of the renderer. However, the animation data is different from graphic data and therefore it should be processed separately by the animation component. The 3D animation component for 3D is conceptually similar to the 2D setup, retrieving animation data from a file in order to modify the skeleton or vertex mesh of a 3D model (i.e. Skeletal vs. Morph Target animation (Gregory, 2009)). Advanced facilities such as blending (transitioning from one animation state to another), inverse kinematics (IK), decompression of animation data, animation playback and procedurally animating free-form body movement are also packaged within this component.

The animation component in our Game Technology Model caters for three variation of animation techniques namely; sprite animation, frame-based skeletal animation and frame-based skeletal procedurally-driven animation. This can be easily extended to include other techniques mentioned earlier. Each of these animation techniques provides a set of methods to manipulate the data defined in the game object (refer to Figure B.9). The core methods include the *setSequence()* method which jumps the animation playhead to the correct sequence and *updateAnimation()* which updates the transform of a mesh or clipping in a sprite sheet. The *RecordAnimation()* method records transform of all objects in the scene graphs into a *playbackBuffer*, whereas the *PlaybackAnimation()* sync with the core update and render method when playback mode is triggered. This declaration act as a wrapper on any form of animation implementation and it is the role of the model translator (MDE tool) to map the choice of game dimension to the compatible animation technique. The detail implementation of this is filled in upon generation of artefacts using the appropriate MDE tool. The construct of the animation component is illustrated in Figure 5.56.

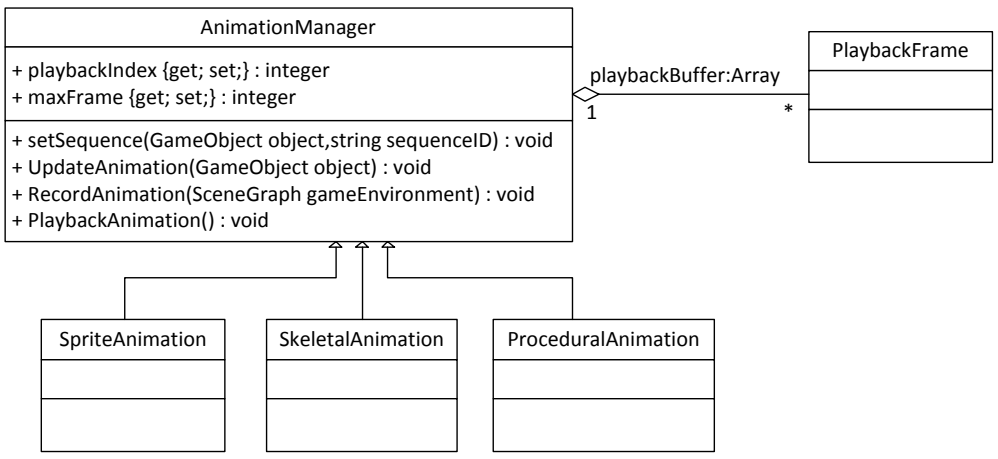

Figure 5.56: Animation Component (AnimationManager) in UML Diagram.

5.4.2.3 Audio

The audio component offers facilities to interface with the audio hardware and manage the playback of audio. In a 3D game engine, the audio subsystem is extended to include 3D audio model which allow game players to perceive sound originating from a positional source.

The audio component in our Game Technology Model consists of a collection of methods that handle the audio task smartly. The assignment of audio channel is managed internally by the component itself. All the complexities of audio playback are encapsulated within the high-level methods such as *Play(), Pause(), Resume()* and *Stop()* featured in Figure B.12.

5.4.2.4 Input

The input component handles all the input events triggered by game players either via traditional human interface devices (HID) such as keyboard, mouse, joystick or newer gestural devices such as the PlayStation® Move and Microsoft® Kinect. Each input event is managed by the input subsystem to trigger specific instructions including GUI events and in-game commands.

The input component in our Game Technology Model listen to all event streams from the hardware interface device. All input interfaces are registered via the *AddListener()* method during initialisation of context (see Figure 5.57). All input event are tracked in the *Update()* method of the individual hardware. Whenever an event occurs, it checks through the *monitoredInputs* list which is maintained by individual hardware and trigger the necessary notification via *Nofity()* method.

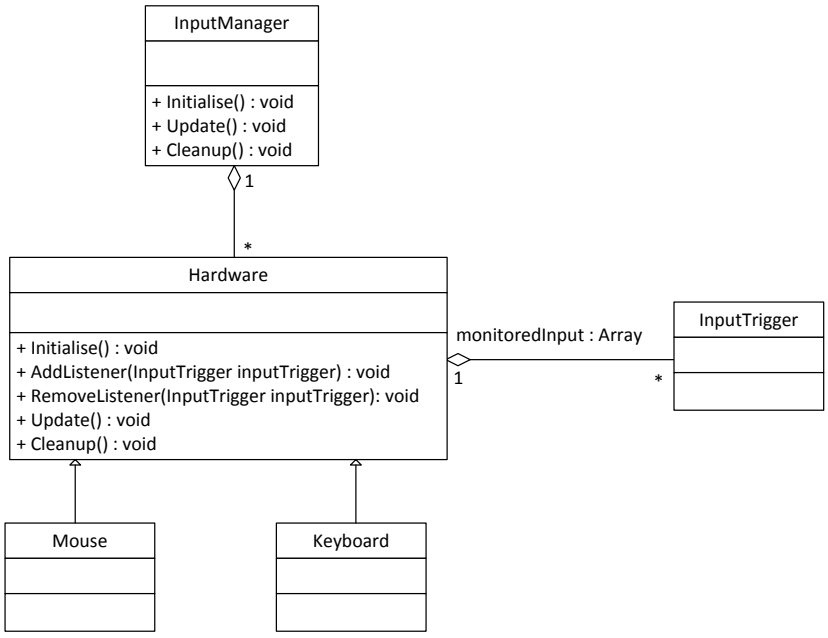

Figure 5.57: Input Component (Input Manager) in UML Diagram.

5.4.2.5 Game Physics

Game physics is the component that applies the law of physics making the game world to behave realistically. Motion, collision detection and collision reaction are the generic forms of computation performed by the game physics component. The complexity and scope of game physics are dependent on game genre and type of game. For example, games such as *Gran Turismo* (http://www.gran-turismo.com/) would expect the physics computation to be very realistic whereas the game physics in *Need for Speed* (http://www.needforspeed.com) is more forgiving and tuned to accommodate the game-play.

The game physics component in our Game Technology Model consist a collection of methods which wraps any game physics implementation into reusable components in our framework (see Figure 5.58). *ApplyForce()* and *GetContsraint()* methods simplify the definition of motion where all the complex computation will be done background through the invocation of *ApplyForce()* method. The constraints in this context refer to the environmental forces which are defined during the initialisation of the scenario through the *RegisterConstraint()* method. Detection of collision is performed via the call of *CheckCollision()* method to check if any two bounding volumes are intersected. When a collision occurs, reaction can be applied. Reaction can be in the form of motion, animation or sound. Reaction in form of motion can be invoked through the *ApplyRecoil()* method.

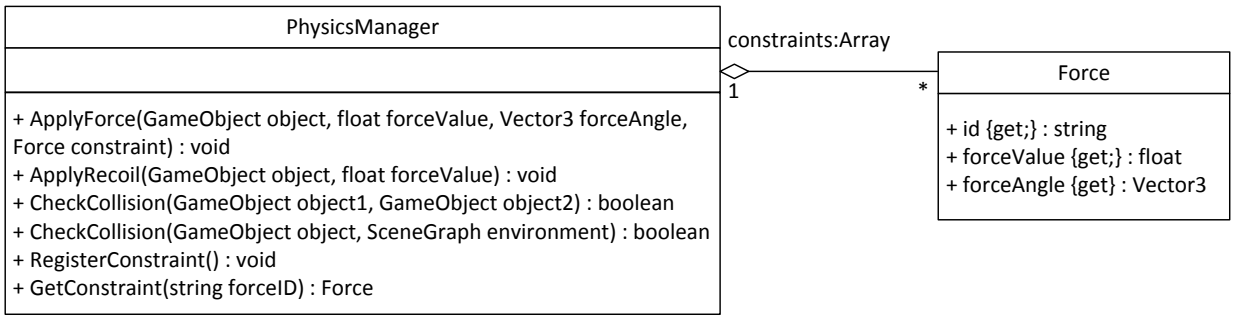

Figure 5.58: Game Physics Component (PhysicsManager) in UML Diagram.

5.4.2.6 User Interfaces (UI): Graphical User Interface (GUI), Media and Heads-Up Display (HUD)

Game software uses GUIs both in-game and out-of-game as a graphical means for accessing functions or commands within the system. Buttons, list-boxes, checkboxs, radio-buttons, textboxs, tabs and scrollbars are each examples of commonly used GUI components. For critical game information that affects game-play, HUD elements such as a digital counters, analogue gauges, mini maps and

horizontal graphical bars are also often used to notify game players about the changes of game state and so aid in the game-playing process.

The GUI components are widely used in both in-game and out-of-game user interfaces. In our Game Technology Model, we listed five different types of commonly GUI components namely button, checkbox, listbox, radiobutton and textbox (see Figure B.14 in Appendix B). In games, GUI can be themed and in-game UI tends to be heavily themed and customised (especially in Real-Time Strategy genre). This can be extended from the five core GUI components defined in our Game Technology Model. Each component has its own data to be updated and specific method to render its representation visually. These implementations are weaved into the Game Technology Model using MDE tools during generation. For optimisation purpose, the GUI components will only be updated at every n^{th} frame which is controlled at the main update loop.

The media components are mostly used during out-of-game setting. Commonly used asset font and sound are pooled in the game resource management component (see Section 5.4.2.8). For less common asset such as image, audio and background sound (which is commonly used in presentation context) (see Figure B.15 for UML representation of media components), these are loaded during initialisation of the context and the states of these components are updated in the *Update()* method which is synced with the parent's update loop. Video and sound components only stores settings and points to the asset location. The playbacks are handled by the Video Player component (see Section 5.4.2.8) and Audio component (see Section 5.4.2.3).

The FED components are in-game GUI designed specifically to provide visual information to game-player in real-time. These highly customised component has the standard *Render()* method and *Update()* method which will have most of the implementation are written. These components will need to be initialised at the beginning of the simulation through the *Initialise()* method where reference variables are established (see Figure B.16 for UML representation of FED components).

5.4.2.7 Video Player

Media such as text, graphics, sound and video are also widely used in games for presenting game related information. The intricate process of video playback is carried out by the video player component. Video playback is often used in game to show cinematic cut-scenes and perform the majority of the storytelling (some games may opt to use scripted animation using the animation component to provide seamless continuity from cinematic to game-play).

The video player component in our Game Technology Model includes the *Play()* method which set up the screen and begins the playback of a compatible video file, *Pause()* method which pauses the video playback, *Resume()* method which resumes the playback, and Stop() method which stops the video playback and notify the *ApplicationEventManager* that the video playback has ended (see Figure B.13). The notification of end of video playback will trigger the transition to next defined context which is defined as part of the flow of the game application.

5.4.2.8 Game Resources Management

Game resources or game assets are usually produced using Digital Content Creation (DCC) tools. These can include 3D models, textures, materials, fonts, 2D sprite sheets, collision data, animation data, sound files, level data and others. The game resource management is the component that provides the facilities for loading and unloading these resources into the game system.

In our Game Technology Model, the Game Resource Management component provides the essential helper methods to load resources and convert the content to the format compatible with the data structures in the Game Technology Model (see Figure 5.59). Shared assets such as fonts and sounds are stored in the respective resource pools for quick retrieval. Each font and sound is tagged when loading into the resource pool for ease of retrieval. Unused font and sound can also be unloaded when it is not used to free up memory.

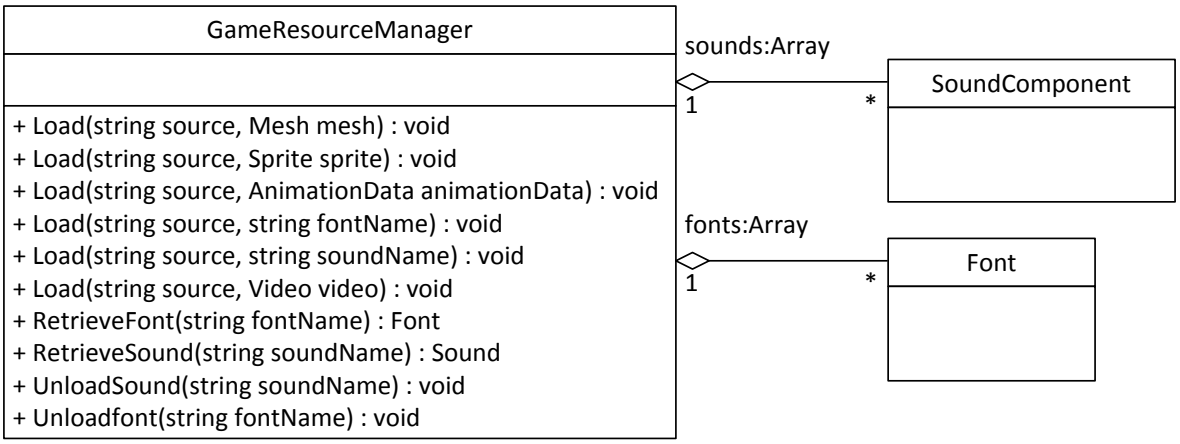

Figure 5.59: Game Resource Management Component (Game Resource Manager) in UML Diagram.

5.4.2.9 Artificial Intelligence (AI)

AI provides Non-Player Character (NPC) abilities to decide, plan and act in a game scenario. Commonly used AI techniques in games include goal-driven decision making, path finding and perception traces which are adjusted to provide a challenging and yet balanced form of game-play.

Again, these techniques can vary depending on games genre. Real-time strategy games such as *Command and Conquer* (http://www.commandandconquer.com/) use a wide range of AI techniques such as chasing and evading, flocking, path-finding and case-based reasoning, whereas games such as *Need for Speed* would rely heavily on waypoint navigation.

In our Game Technology Model, the AI component is designed to be integrated with the game logic but also extensible. Each technique is treated as an instance to process specific data of a game object through referencing. Whenever a specific AI technique is required, it is accessed directly via the instance of the technique. The AI component's *Update()* method is synchronised with main update method to ensure game objects are updated at each frame. This is invokes the update method of the AI techniques such as path finding and flocking shown in Figure 5.60.

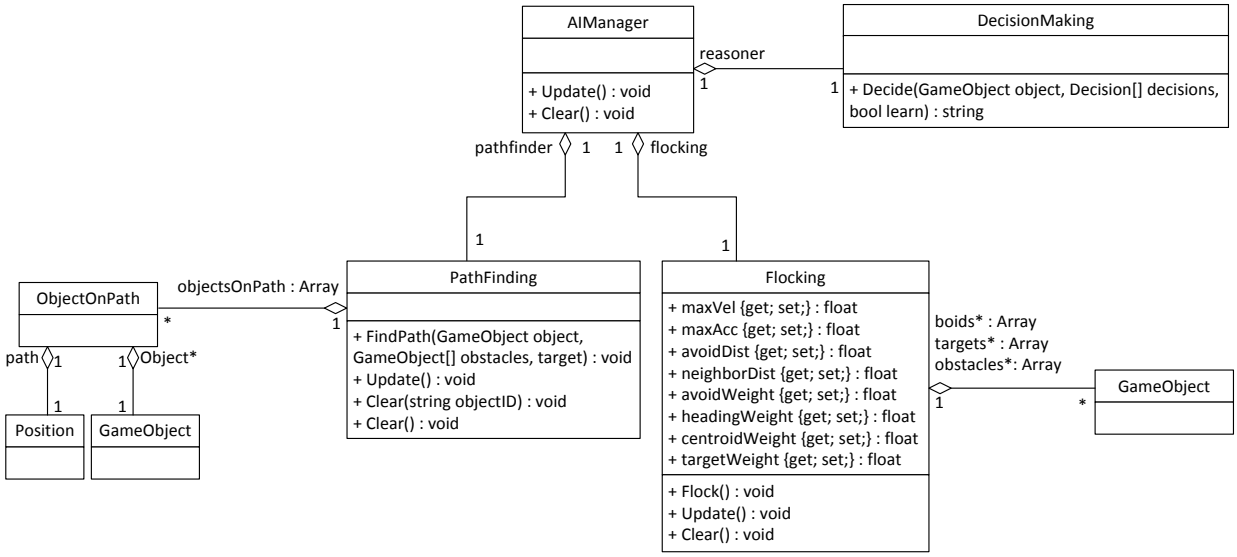

Figure 5.60: AI Component (AIManager) in Class Diagram.

5.4.2.10 Networking

The networking component is used in games software to facilitate multi-players. It handles all the aspects of communication between client and server. Elements of networking component include authorisation, authentication, structuring and filtration of game messages, low-level communications, task interface, protocols, network introspection, and data interpolation and extrapolation (Carter, Rhalibi, Merabti, & Bendiab, 2010). This component is excluded from use when game software is developed solely for single player on local machine.

5.4.3 Helper Components

Other components which are commonly used and deem useful in game software includes the math library, random number generator and unique object identifier management.

5.4.3.1 Math Library

The math library provides the base classes for representing game objects transforms for virtual positioning and bounding volume for a range of application including collision detection, AI, etc. In our Game Technology Model, we identified the four most used math-based classes as Vector2, Vector3, Vector4 and Matrix. These are featured in Figure B.17.

In addition, it also provides the necessary arithmetic methods for computing these data structures. The arithmetic methods for vectors will include computing the length of the vector; computing the square length of the vector; dot product between two vectors; cross product between two vectors; sum of two vectors; difference of two vectors, scaling of a vector; interpolate between two vectors; normalisation of a vector; transforming a vector or an array of vectors, it's coordinate and it's normal using given a matrix; and, projects and unprojects a given vector3 or an array of vector3s from object space to screen space.

The math library should also include a range of methods for computing matrix such as the identity matrix; finding the determinant of a given matrix; decomposing a matrix into scale, rotation and translation vectors; computes the transpose of a given matrix; computes the product of two given matrices; computes the inverse of a matrix; building matrix for scaling, translation, rotation, transformation (both 3D and on 2D (xy-plane)), affine transformation (both 3D and on 2D(xy-plane)), look-at (camera focusing on a focal point), perspective and orthographic.

The math library should also include commonly used math constants such as PI (π) and trigonometry functions sine, cosine, tangent, cosecant, secant and cotangent for calculating angles. Other functions that may be of great help also include methods for converting between radian and degree, value rounding and truncation, and physics-related methods such as building bounding volumes (box and sphere) and checking of intersection between a ray and a volume.

5.4.3.2 Random Number Generator

Random number generator is a "must-have" facility in game software. It is used in dynamic generation of game objects, varying factors that create dynamic conditions in scenario, decision making in AI etc. Most random number generators are in fact pseudo random number generator

which produces a sequence of numbers which appears to be random in a repeated cycle. A seed (usually system time) is passed in to initialise the starting point of the sequence by means of disrupting the predictability. However, modern software platform may have seeding integrated in the generator which makes implementation cleaner. In terms of outputs of the random number generator, some produce floating point number and others generate integer. In our Game Technology Model, we choose to have a random number generator with integrated seed that outputs both floating point and integer value with some helper methods such as generating random number between a given ranges of value (see Figure B.18).

5.4.3.3 Unique Object Identifier Management

A large quantity of game objects are created during runtime in any game scenario to enliven the scenario and make game-playing more dynamic. Each of the game objects created need to have a unique identifier for identification and referencing purposes. The unique object identifier management is the helper component that generates a unique identifier. In our Game Technology Model, this helper component returns an identifier that begins with a prefix and followed by integer. The prefix allows classification of game objects and it is registered during initialisation of a game scenario. This component is illustrated in Figure B.19.

5.5 Game Software Model (GSM)

The Game Software Model is the final layer of model in our model-driven framework, which promotes interoperability of game software. It completes the representation of computer game software by translating and transforming the Game Technology Model to specific technology platform. The Game Software Model for each technology platform varies from one another and this is defined by someone who knows the targeted technology platform inside-out and understands the model-driven approach proposed in this thesis.

The Game Software Model is designed by a technical person who possesses great understanding of the model driven framework and the technology platform. Developers of the Game Software Model may choose one of the two different perspectives; (1) to bridge the Game Technology Model to an existing game software framework (e.g. Microsoft's DirectX) or (2) to implement game software from scratch for an intended technology platform. Both these exercises may require platform specific details or components to be added which has been omitted in the Game Technology Model. It is worthy to note that Game Software Model is incrementally built-up from Game Technology Model with the

inclusion of the aforementioned platform specific components (see Figure 5.61 and Figure 5.62 for an overview of Game Software Model). In this section, we can only provide necessary guidelines for defining the Game Software Model for the two approaches identified as it is not feasible in the current scope of this project.

5.5.1 Mapping Game Technology Model to a Specified Game Software Framework

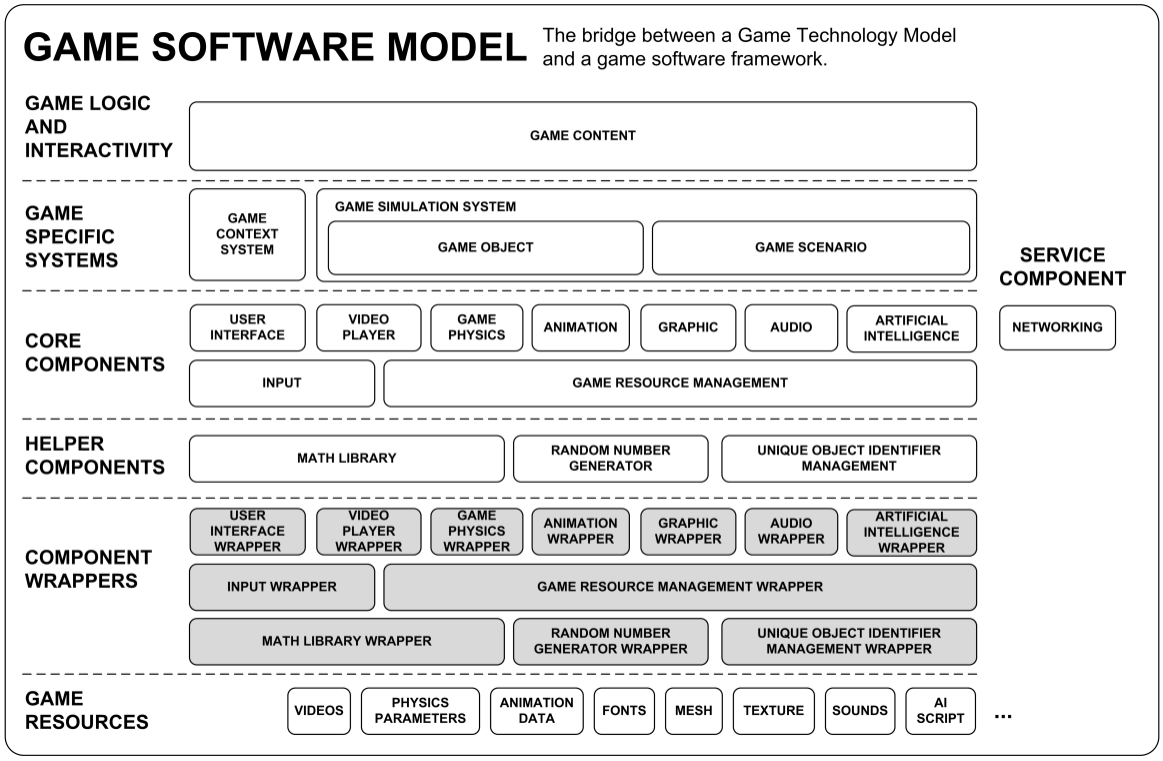

Figure 5.61: Overview of Game Software Model that bridge Game Technology Model to a game software framework.

Implementing the Game Software Model for an existing game software platform will require framework developers to map the components presented in the Game Technology Model to the chosen game software framework. Often it is likely to achieve a one-to-one mapping of Game Technology Model components with game software framework components with possible inclusion of information that is required by the game software framework. In some cases, it may require Game Software Model designer to provide additional constructs to work around the game software framework in order to components from Game Technology Model mapped across. Details of the additional information and constructs would depend on the choice of game software framework and how the Game Software Model designer maps Game Technology Model to the chosen game software framework. Figure 5.61 illustrates the overview of Game Software Model with component wrappers

(shaded in grey colour) which map components of Game Technology Model to the appropriate component in a game software framework.

5.5.2 Additional Platform Specific Details to Game Technology Model for a Software Technology Platform

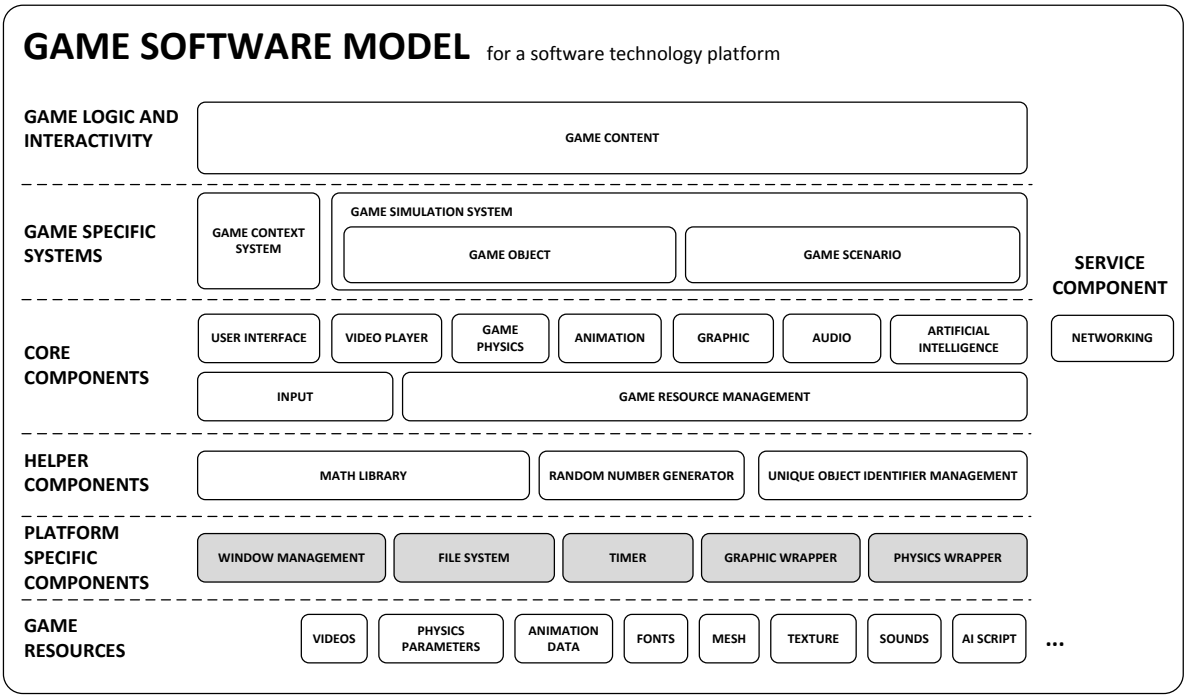

Figure 5.62: Overview of Game Software Model that includes platform specific components for a software technology platform.

Unlike the former approach described in Section 5.5.1, designing a Game Software Model for a specific software technology platform will require the addition of certain platform specific components which are used by the core components. These are identified by Gregory (2009) as *window management*, *file system*, *timer*, *graphics wrapper* and *physics wrapper*. Most game software frameworks would have these platform specific components implemented in low-level code that is coupled to a specific operating platform to ensure it delivers the performance required of the game software. In our approach, these platform specific components have been omitted from the Game Technology Model and are only added in the final stage of the model transformation to achieve true operating platform independence. Implementers of our Game Software Model will have to define these platform specific components so they can be mapped to the right implementation during the generation of program code. This makes the Game Software Model structure differ from the earlier version described in Section 5.5.1 as the component wrapper is replaced with platform specific components (shaded grey in Figure 5.62). In the following subsections, we describe the functional

requirements of these platform specific components as guidelines for Game Software Model designers to design the Game Software Model for the platform of their choice.

5.5.2.1 Windows Management

The window management component handles the creation of windows (canvas areas) for displaying game content visually on screen in a game application. Windows are layered on top of each other with the active window being on top. In our Game Technology Model, the ordering of windows is achieved using an active context stacks as explained in Section 5.4.1.1. This window management component serves as a foundation to the game context system and provides the facilities to ease the creation of windows to display game content in different context. Functional requirement on this component is shown in Table 5.2. This component should work side-by-side with the Game Context Manager described in Section 5.4.1.1.

Table 5.2: Functional Requirement for Window Management Component

Functional requirement(s)
Create a window for displaying context based on given dimension, position and other parameters such as title of the window, setting focus to this window, allowing or disallowing window to be minimised and framing of the window.
Minimise a window.
Restoring a window to its original dimension.
Removing a window from the application.
Retrieving context from a window.

5.5.2.2 File System

File system is very dependent on the operating platform. Each operating platform has its own meticulous approach of storing and retrieving a file. In the context of game software, file system is the basic facility for loading game resources by the Game Resource Manager, game settings and game player's progress from file into the game application. The saving facility allows the application to create files to store game player's progress and amended application setting on the operating platform for later retrieval. Functional requirements for this component are described in Table 5.3.

Table 5.3: Functional Requirement for File System Component

Functional requirement(s)
Read binary file from a given file path
Read text file from a given file path
Write binary file to a specified file path
Write text file to a specified file path

5.5.2.3 Timer

The timer component is crucial for any real-time software such as game as it provides the notion of time which is used in computing time-step and other aspects in game software such as motion, animation and recording player's performance. In our Game Technology Model, we have opted for fixed time-step game loop which rules out the use of timer in motion and animation. Nevertheless, timer component still plays a major part in computing fixed time-step and recording performance of game-player. The timer component only has one and only one important facility; to query the processor's timer. For finer time calculation, some timer provides facilities to access processor's cycle and frequency to compose a high-resolution timer (nanoseconds). In our Game Technology Model, we will only require milliseconds timer. This is described in the functional requirement in Table 5.4.

Table 5.4: Functional Requirement for Timer Component

Functional requirement(s)
Read CPU time in miliseconds

5.5.2.4 Graphic Wrapper

The graphic wrapper encapsulates functionalities of graphic library into reusable methods for use in game development. DirectX, Java 3D, OpenGL and OpenGL ES (for embedded systems such as smartphone, tablet and portable game consoles) are examples of graphic library available for selection. Selection of the graphic library is dependent on the choice of software technology platform. For example, the DirectX graphic library is only compatible with software technology platform just as the native C/C++ and the .NET framework. So for someone who wishes to develop on Java, they will have to choose Java 3D or OpenGL graphic library. In our Game Technology Model, the graphic wrapper should provide the necessary functionalities as described in the functional requirement in Table 5.5 which are used by renderer component described in Section 5.4.2.1.

Table 5.5: Functional Requirement for Graphic Wrapper

Functional requirement(s)
Interface with the hardware device
Access the buffers
Perform transformation to the vertices
Rasterize vertex into pixel
Render to screen
Render to buffer
Merge render output to form a final output

5.5.2.5 Physics Wrapper

The physics wrapper, like the graphic wrapper, encapsulates functionality of a 3rd party physics library for use in the game software. There is a range of physics library available for use such as Box2D, Open Dynamics Engine and Havoc Physics are few to name. Primarily, these physics engine provides all the functionalities to compute motion, trajectory and detect collisions. The aim of the physics wrapper is to provide the interface to these functionalities so it can be used by the game physic component described in Section 5.4.2.5. The functional requirements of the physics wrapper are described in Table 5.6.

Table 5.6: Functional Requirement for Physics Wrapper

Functional requirement(s)
Computing motion using force and constraints
Computing recoil based on given force and transform

5.6 Chapter Summary

In this chapter, we have described our novel model-driven computer game framework detailing our architectural strategy, the framework itself and the models namely Game Content Model, Game Technology Model and Game Software Model

Our novel model-driven game framework is designed to support the production of a variant of computer game software that covers a wide range of technology platform as well as operating platform. The loose-coupling of modules (as described in Section 5.2) provides developers the flexibility to substitute modules and yet maintain the integrity of relationships between the modules. At the core of the framework are the models namely the Game Content Model, Game Technology Model and Game Software Model.

Our novel Game Content Model improves on the existing work GOP, RAM and NESI by providing a formalised approach and a complete set of concepts for representing game design. It combines our study on game design, game development and computer game with a selection of concepts from the aforementioned existing works to form a formal model. We have shown how VSM can be exploited for use in designing the game object concept as described in Section 5.3.4. The VSM has been used as a tool to enable us to identify the key attributes to representing a game object in our Game Content Model. Our Game Content Model is unique and complete formalised model for representing game design amongst with the game models described in Chapter 3.

Our Game Technology Model is designed based on the data-driven architecture and include the essential game specific systems and core components of software which facilitates the operation of the computer games. The Game Context System handles the transitions between contexts and work cooperatively with the Game Simulation System to simulate a scenario. For ease of processing, scene graph is used to organise render-able and updatable objects such as media components, GUI components, FED components, game objects and lights. These objects are processed using the platform independent core components such as renderer, animation, audio, input, game physics, user interfaces, video player, game resource manager, networking and artificial intelligence. Supporting these core components are the helper components namely the math library, random number generator and unique object identifier management. The architecture of our Game Technology Model is designed to allow mappings between the components of the Game Technology Model and the components of any other game software framework possible. This common setup of our Game Technology Model can be used by developers to create their own proprietary game software framework. Alternatively, it can be regarded as a generic virtual wrapper for existing game software frameworks. The functionality of each component defined in the Game Technology Model act as interfaces that wrap a different implementation of a game technology. This allows computer games software to be produced on different technology platforms through code generation which reads the Game Technology Model and translates it into software artefacts.

The Game Software Model is the platform specific software representation of the game software. It is a Game Technology Model with the inclusion of platform specific information or components which has been excluded from the Game Technology Model. Game Software Model is designed by a technical person who possesses great understanding of the model driven framework and the technology platform. Due to the wide range of technology platform available, we chose to describe the

platform specific details to be added to the Game Technology Model based on the approach chosen. These platform specific components can be replaced without affecting much of the other aspects of the software defined in the Game Technology Model when a new operating platform is chosen. Finally, the Game Software Model can then be interpreted using MDE tools to generate the necessary software artefacts including software codes and software build.

In summary, our novel model-driven game framework could be a blueprint for framework developers who wish to develop high-level tool using model-driven approach for non-technical domain experts to produce a range of computer games for use in game-based learning. The Game Content Model provides the building blocks for designing a variation of computer games while the Game Technology Model and Game Software Model provide the software constructs that make the computer game functional as game software on the targeted operating platform. Although we have gone through a lot of detail describing the technical aspects of our model driven framework, it should be borne in mind that the model driven engineering approach hides all of these from the actual domain experts and we include these only for completeness. In next chapter, we present a case study to demonstrate the production of serious games using our model-driven approach and evaluate the works carried out in this thesis.

CHAPTER 6 – CASE STUDY & EVALUATION

This chapter presents a case study on the use of the modelling environment and MDE tools developed using our Model Driven Games Development Framework described in Chapter 5 and a critical evaluation of the work carried out in this research. Our case study describes how a domain expert would model a serious game and use our Serious Games Modelling Environment (SeGMEnt) and our MDE tools to produce a serious game. A comparison with conventional production approach is then made. The critical evaluation covers an evaluation of the framework proposed in Chapter 5, the prototype implemented in Appendix C and the case study conducted in this chapter.

6.1 Case Study: An application to Serious Games for Game-based Learning

This research project was set out to design a model driven framework that supports the development of serious games for use in game-based learning. One of the challenges this research study aimed to address was to provide domain experts with few game development skills a tool which they can use to develop or prototype serious games. The SeGMEnt and the MDE tools are designed specifically for this purpose. In this case study, we demonstrate a use case scenario of our model driven approach and show how it can help non-technical domain experts in the production of serious games in comparison to the conventional approach of serious game production.

For our case study, we approached a volunteer teacher at Christian Fellowship School in Liverpool and explained to her about game-based learning and the benefits of this approach. Prior to becoming a volunteer teacher, she was a lecturer teaching general computing subjects in a college. We asked if she would consider using a game-based learning approach to teach her students a specific topic. After discussion, we reached an agreement to produce a serious game for educating students from age seven to nine about fire safety and evacuation procedures in a classroom setting. In the following sections, we describe the process of designing the serious game followed by a walkthrough of modelling using our SeGMEnt and MDE tools before we present the findings and our analysis in Section 6.1.2.

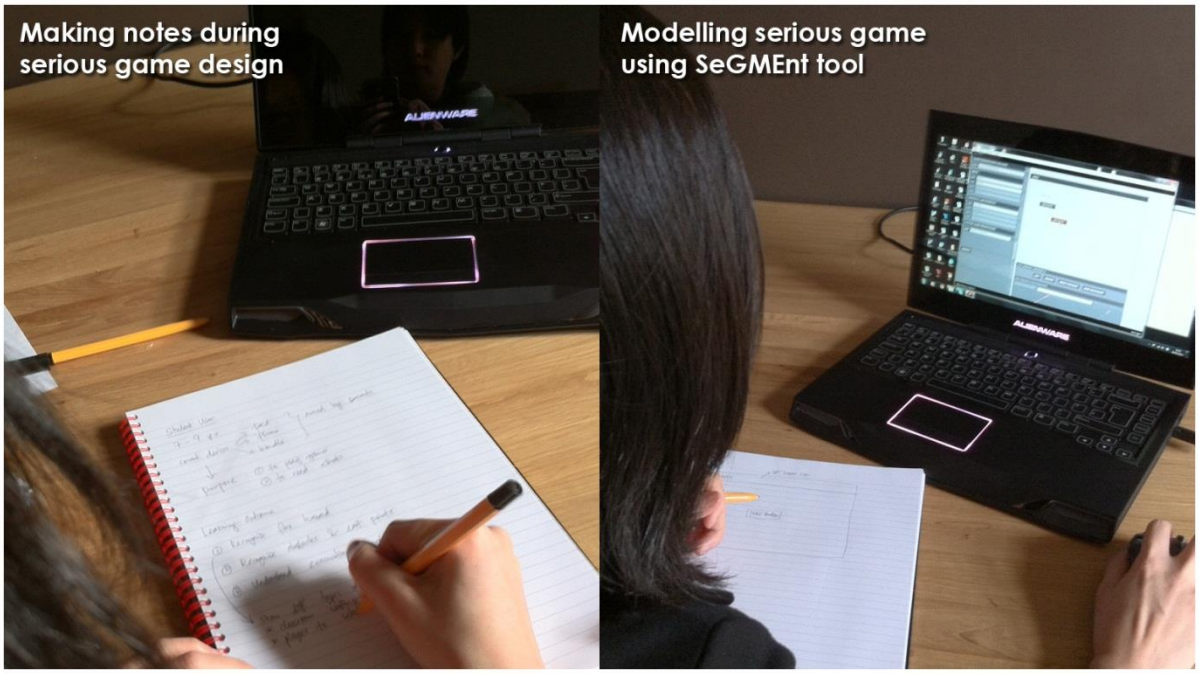

Figure 6.1: Case Study in progress

6.1.1 Serious Game Design Process

Designing serious games differs greatly from conventional game design process because it principally involves pedagogy. This requires a hybrid game design methodology that infuses activities from instructional design beginning with the definition of learning objectives. The methodology should allow a close collaboration between domain experts and the game design team during the process of serious game design and often requires a number of iterations (involving various stages of design, rapid prototyping, play-testing and revision) before it is released for use (H. Kelly et al., 2007). These measures are taken to ensure that actual learning takes place within serious games. The serious game design methodology shown in Figure 6.2, from our previous work in (Tang & Hanneghan, 2010a), is used in this case study as a method to design the serious game. This case study only covers the *planning* phase and the *prototyping* phase (one-iteration). The *finalising* phase of the methodology is omitted because it is not within the scope of this case study.

For the benefit of the subject, we explained and walked through the serious game design process with the subject since she is not familiar and has no experience with serious games design and this is common amongst new practitioners of game based learning. Throughout this process, all the designs are documented on paper and there are no software tools being used at this stage.

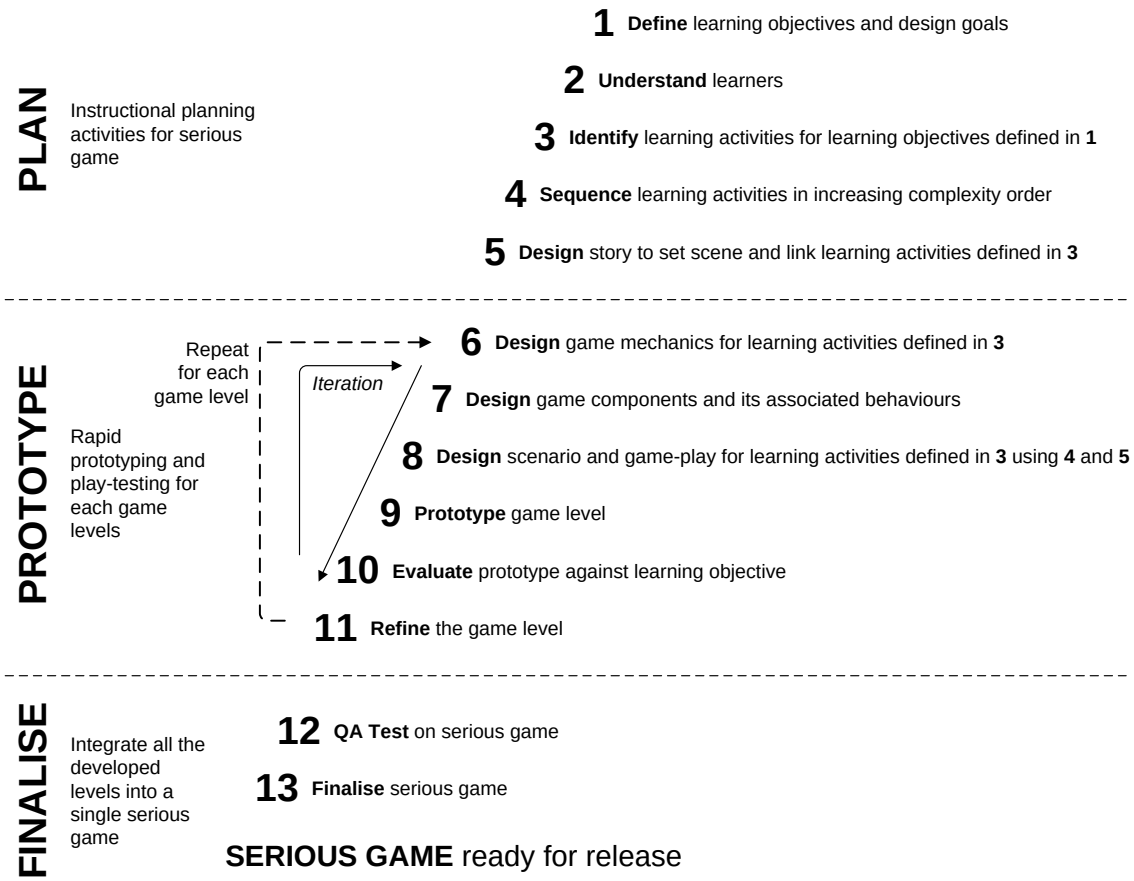

Figure 6.2: Serious Game Design Methodology (Tang & Hanneghan, 2010a).

We begin the serious game design process by defining the learning objective and design goal. This task is not new to teachers as it is a crucial task in lesson design. The learning objectives defined for the fire safety and evacuation procedure are shown in Table 6.1.

Table 6. 1: Learning Outcomes for Fire Safety and Evacuation Procedure

Upon completing the serious game, game players will learn
To recognise fire hazards,
To recognise obstacles in a classroom,
To recognise exit points,
To be clear of the evacuation procedure in a classroom.

The next task is to understand the learners' (game players') needs in the serious game so we can choose the right choice of interaction model and theme for the audio-visual material. The only details we are certain about the learners at the moment is that they are studying in Year 1 (7 years old) to Year 3 (9 years old). The approach we chose to use was persona analysis (Nielsen, 2013). With our

help and experience from the subject, the subject has identified three behavioural personas to represent the learners (see Table 6.2). These personas indicate that students are exposed to interactive digital content on smartphones and tablets. Analysing these personas, we concluded that the interactions have to be simple, content has to be engaging and short in length, and finally learners should be able to use the applications without much instruction.

Table 6.2: Personas for Students from Year 1 (7 years old) to Year 3 (9 years old) in Christian Fellowship School

Name	**Matthew**	**Jimmy**	**Sandy**
Gender	Male	Male	Female
Year	Year 1	Year 2	Year 3
Age	7	8	9
Activity	"I do use my dad's iPhone to play Temple Runs!"	"I play Fruit Ninja game on iPad and they are nice!"	"My mom lets me read my storybooks on her Kindle".
Devices used	iPhone	iPad	Kindle Fire

The next task involved the identification of learning activities that should be included in the serious game. These learning activities should be meaningful activities and should allow game-players to relate directly to real-life objects, scenario or matters. A brainstorming session took place and was followed by a discussion to decide on the most appropriate learning activities that meet the learning outcomes. The learning activities which were identified to be most suitable are presented in Table 6.3.

Table 6.3: Learning Activities for Fire Safety and Evacuation Procedure serious game

Learning Outcomes	**Learning Activities**
To recognise fire hazard,	Different types of fire hazard are set around a classroom and the game player has to identify them before a fire incident.
To recognise obstacles and exit points in a classroom,	A classroom setup with different obstacles obstructing the path to the exit points. The game player has to identify the obstacles and clear the path to exit the classroom.
To be clear of the evacuation procedure in a classroom.	A fire incident has occurred somewhere in the school and the game player has to take the role of the teacher to direct all the children to the exit point in a given duration of time without risking the safety of the children.

These learning activities are then ordered in an increasing order of difficulty. This is to ensure that pre-requite learning is achieved before higher level challenges are presented. The ordered activities are presented in Table 6.4.

Table 6.4: Learning Activities ordered according to increasing difficulty level

Order of Difficulty	Learning Activities
1	Different types of fire hazard are set around a classroom and game player has to identify them before a fire incident.
1	A classroom setup with different obstacles obstructing the path to the exit points. Game player has to identify the obstacles and clear the path to exit the classroom.
2	A fire incident has occurred somewhere in the school and game player has to take the role of the teacher to direct all the children to the exit point in a given duration of time without risking safety of the children.

Once the learning activities have been ordered, a story is created to motivate the learners to participate in the goal-directed learning activities. After some brief discussions and advice on game design from us, the subject decided to have a story that feature the fire incident at a fictitious school and the ending of the story would depend on the game player's achievement. The learning activities around identification of obstacles, exit points and fire hazards are a series of drills that the fictitious school conducts at the beginning of the academic year. The game player is required to complete all the drills and achieve a satisfactory score (this is subject to game tuning in later state of the prototyping) before the game progress with different fire incidents throughout the term. Upon completion of all challenges, the story ends with the school being awarded as "fire safety award of the year". If game-players failed in any of the scenarios involving a fire incident, the story would end with the school on the news with the headline of "Fire Tragedy at School Killed 30 School Children" and followed by messages about the importance of the fire drills and prevention. The flow of the story and play spaces are illustrated in the Figure 6.3.

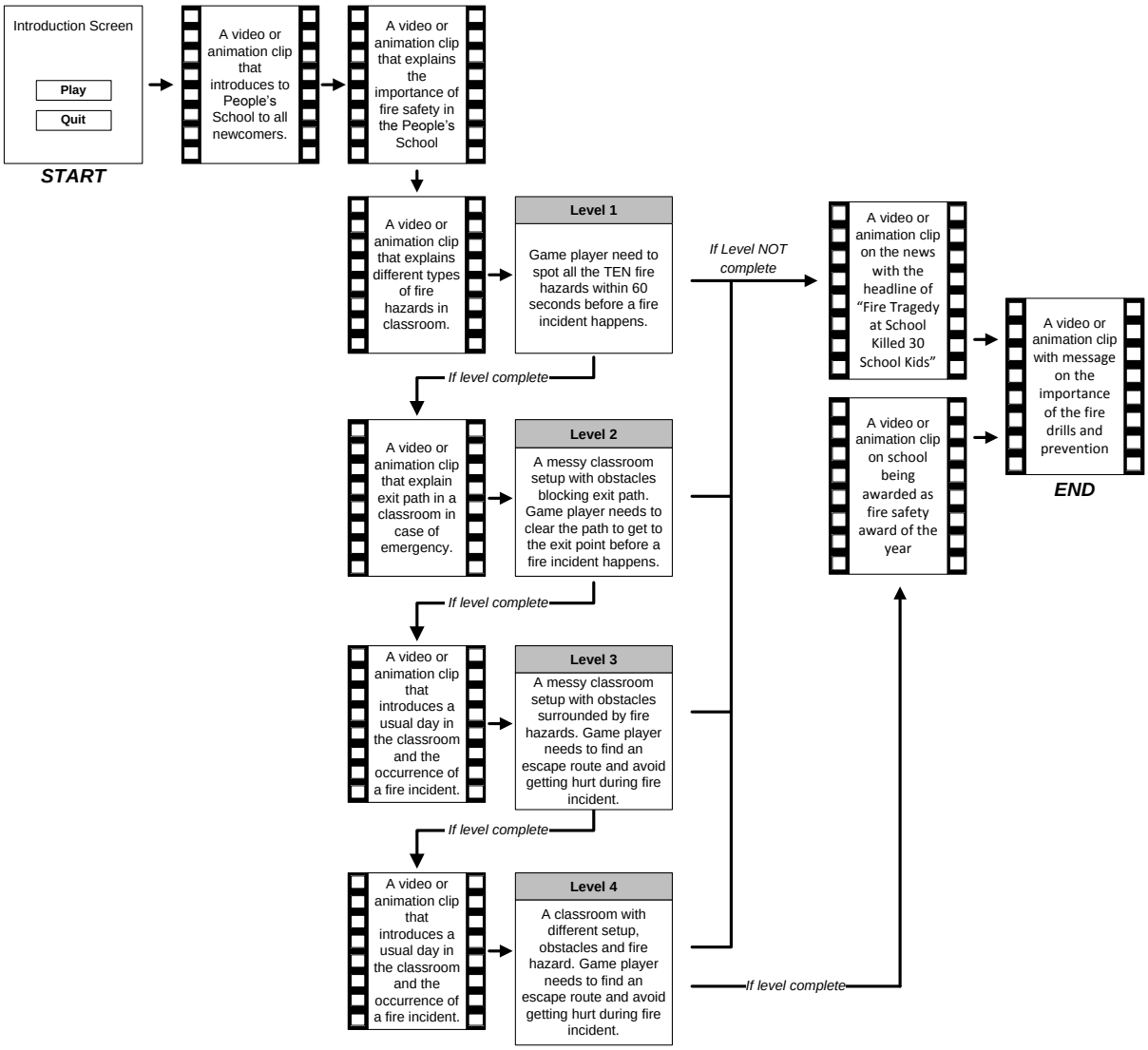

Figure 6.3: Flow of story and play spaces within the story for the Fire Safety and Evacuation Procedure serious game reproduced based on sketches from the case study conducted

This is then followed by the design of game mechanics and game components for the learning activities defined. The design of game mechanics and identification of game components are done with our consultation due the unfamiliarity of subject on game design and these are shown in Table 6.5.

Table 6.5: Game mechanics and game components for different learning activities.

Learning Activities	Game Mechanics	Game Components
Level 1	**Interaction Model:** Mouse Click **Game Objective:** Spot 10 fire hazards within 60 seconds. **Game Events:** Fire will start after 60 seconds game has started. **Scoring:** Each fire hazard identified will be awarded with 100 points. Player will be awarded with 10 bonus points for each second remaining in the countdown clock. **Game Stopper:** All fire hazards have been identified or 60 seconds countdown has lapsed. **Progression:** Progress to Level 1 if level objective is achieved.	**Static components:** Background image **FED Components:** Timer, Score & Remaining item counter **In-game object:** 10 fire hazards, combustible items such as tables, chairs etc.
Level 2	**Interaction Model:** Mouse Click **Game Objective:** Spot 10 obstacles within 60 seconds. **Game Events:** Fire will start after 60 seconds game has started. **Scoring:** Each obstacle identified will be awarded with 100 points. Player will be awarded with 10 bonus points for each second remaining in the countdown clock. **Game Stopper:** All obstacles have been identified or 60 seconds countdown has lapsed. **Progression:** Progress to Level 3 if level objective is achieved.	**Static components:** Background image **FED Components:** Timer, Score & Remaining item counter **In-game object:** 10 obstacles such as tables, chairs, bins etc.
Level 3	**Interaction Model:** Mouse Click to move obstacle and control navigation of avatar. **Game Events:** Fire incident will start 10-15 seconds after game has started. Fire will spread to cover the whole area in 40 seconds time. **Scoring:** Each fire hazard identified will be awarded with 100 points. Player will be awarded with 10 bonus points for each second remaining in the countdown clock. Each encounter with fire will cost 50 units of health. **Game Stopper:** Player's health unit reaches zero or player reached exit point **Progression:** Progress to Level 4 if level objective is achieved.	**Static components:** Background image **FED Components:** Timer, Score & Health bar **In-game object:** 3 obstacles such as tables, chairs, bins etc.
Level 4	**Interaction Model:** Mouse Click to move obstacle and control navigation of avatar. **Game Events:** Fire incident will start 10-15 seconds after game has started. Fire will spread to cover the whole area in 30 seconds time. **Scoring:** Each fire hazard identified will be awarded with 100 points. Player will be awarded with 10 bonus points for each second remaining in the countdown clock. Each encounter with fire will cost 50 units of health. **Game Stopper:** Player's health unit reaches zero or player reached exit point **Progression:** End of game.	**Static components:** Background image **FED Components:** Timer, Score & Health bar **In-game object:** 5 obstacles such as tables, chairs, bins etc.

6.1.2 Modelling Serious Game in SeGMEnt

Once the game mechanics and game components have been decided, we can then model the serious game in the SeGMEnt tool. Modelling in SeGMEnt follows a bottom-up approach which requires

modeller to model basic elements in order to model more complex elements which are composed of the basic elements. For example, in order to model a game scenario, the modeller will first have to model the game object. In this section, we describe the serious game modelling process in SeGMent which the subject has gone through (see Figure 6.1). For the brevity of this thesis, we will only present the modelling of "Level 1" which requires game player to spot all the ten fire hazard game objects within 60 seconds before a fire incident happens in detail and the introduction screen for the serious game.

There are seven successive stages to follow when modelling a serious game in SeGMEnt (see Figure 6.4). The first stage in modelling serious game in SeGMEnt involves modelling of game object. This is done using the game object viewpoint as described in Section 0. According to Burnside (2008), objects which could be a potential source a of fire outbreak such as halogen bulb, candles, light decoration and easily combustible such as papers, books and thin paper-based decorations are considered hazardous in a classroom setting. The subject has decided to place halogen bulb, candles, stack of papers, workbooks and paper decorations strategically in different location of the classroom. The combustible objects are placed closely to halogen bulbs and candles. Other game objects included in the classroom setting includes students, chairs, tables, shelves, whiteboard and blinds. Subject then defines these game objects in the game object viewpoint and used placeholder images (which were created by us during the case study using Adobe Photoshop as external files) as visuals for each of the game objects (see Figure 6.6).

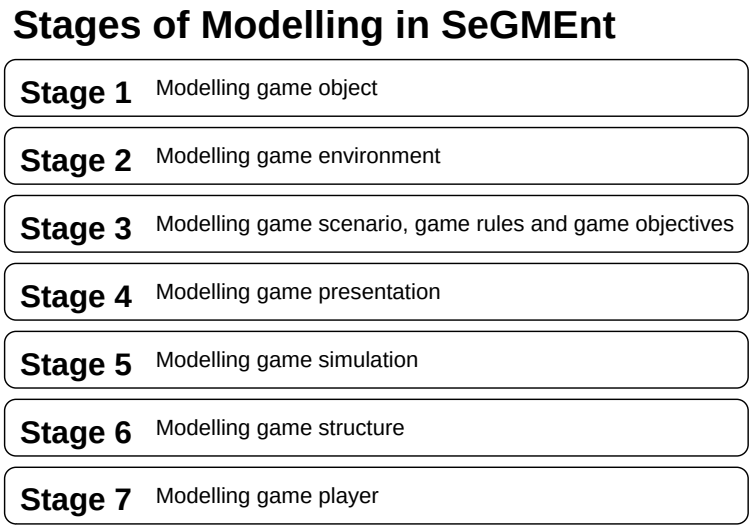

Figure 6.4: Stages of modelling in SeGMEnt

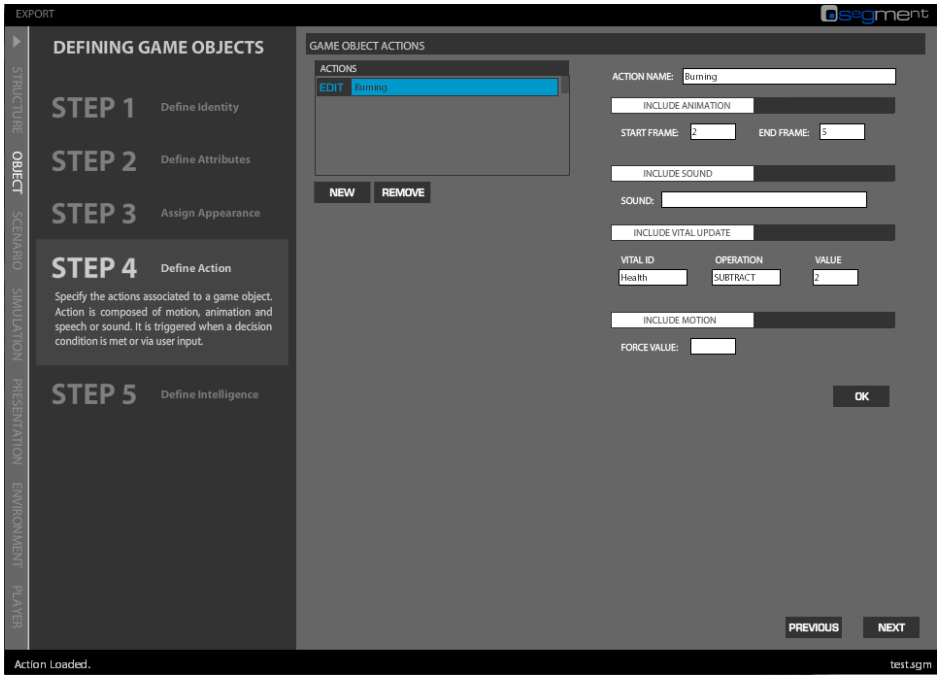

Figure 6.5: Screenshot of the subject defining game object action named “burning” in game object designer viewpoint.

Once all the game objects were defined, the subject composed the classroom setting for “Level 1” with the game objects defined earlier using the game environment designer viewpoint. This involved positioning of the game objects in the game environment. Screenshot in Figure 6.6 shows the completed environment named “Classroom1” composed by the subject.

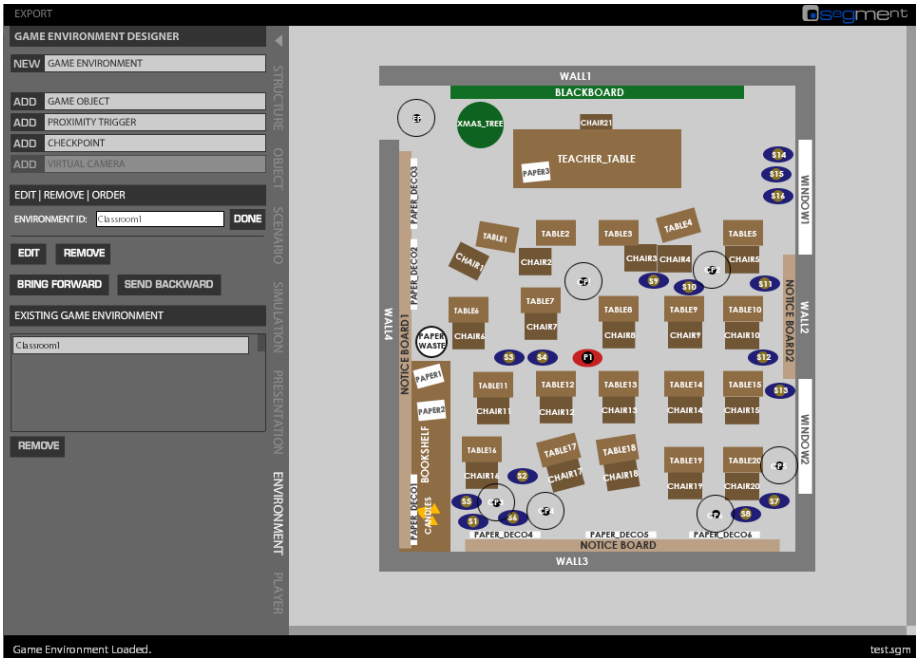

Figure 6.6: Screenshot of the completed game environment modelled by the subject.

The next stage involved the modelling of game scenario where the subject defined the game events, game objectives and game rules for each scenario in the serious game using the game scenario design viewpoint. The "Level 1" scenario consists of events where students (game objects) are animated. After 60 seconds an event of fire will be triggered where the paper decoration nearby the bookshelf catches fire from the candles placed near. This event is followed by game objects such as chairs, tables, shelves, stack of papers, decoration, catching fire. 10 seconds after the fire event, students (game objects) will animate and move towards the exit point. The fire will spread around the classroom and exit point before the remaining three students (game objects) can escape. See Figure 6.7 and Figure 6.8 for the modelling of these serious games aspects in game scenario designer viewpoint in our SeGMEnt tool.

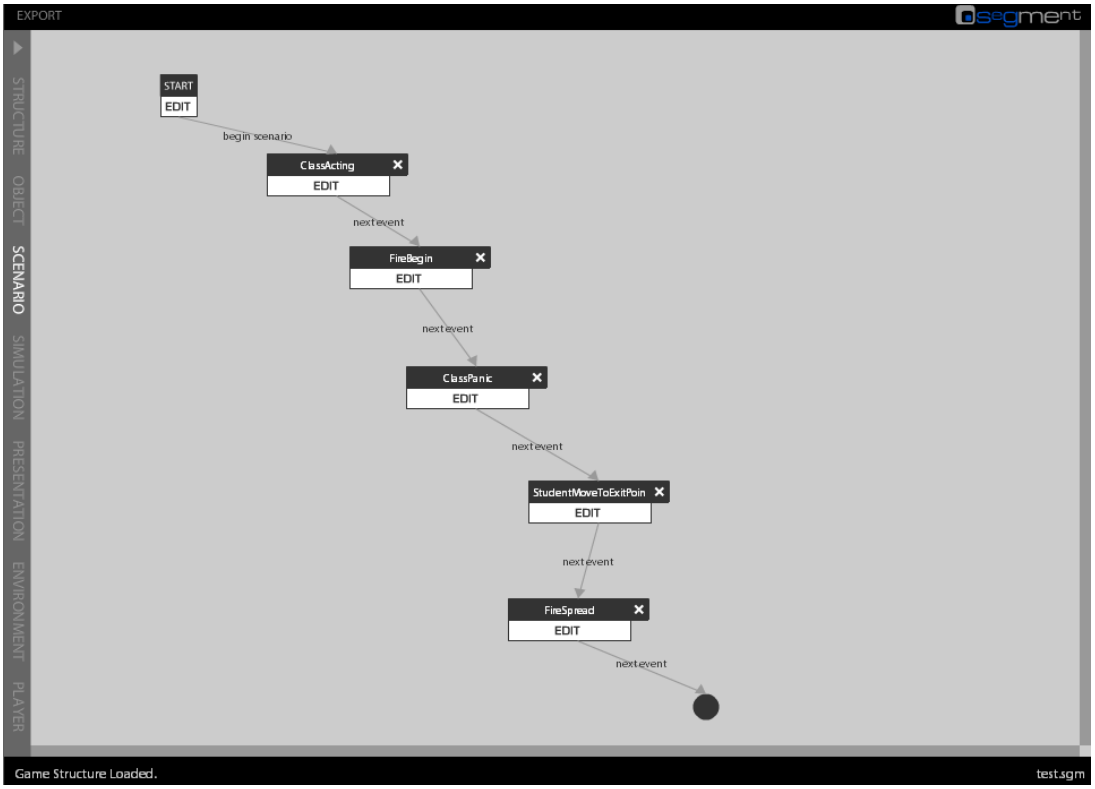

Figure 6.7: Screenshot the completed game event modelled by the subject in the game scenario designer viewpoint.

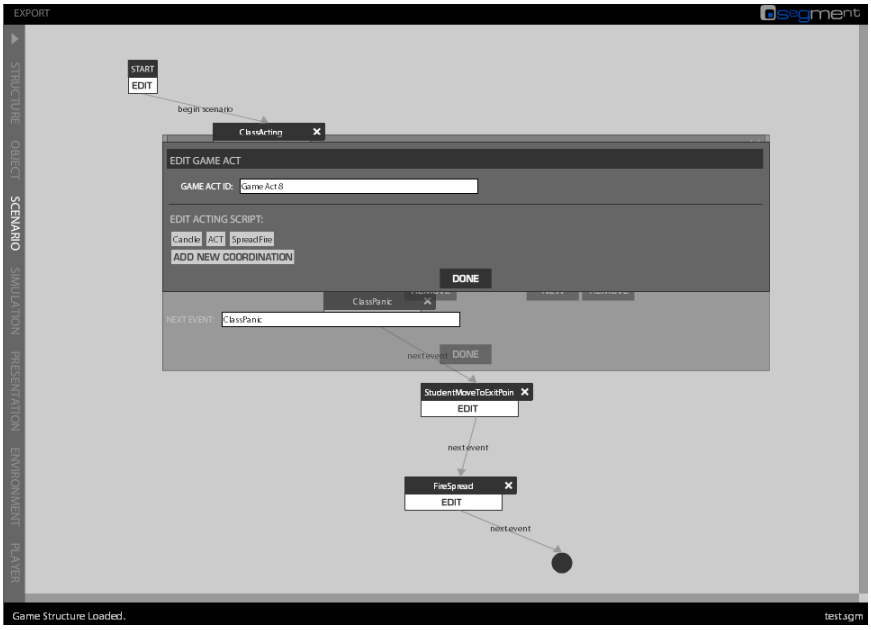

Figure 6.8: Screenshot the ActingScript composed by the subject in the game scenario designer viewpoint.

In terms of game objectives for the scenario "Level 1", game players will have to identify all the stacks of paper (game objects), paper decorations (game objects) and waste paper basket (game object). Each object will consist of a state named "removed" with initial value of 1 and when interact with will have the value changed to 0 (zero).

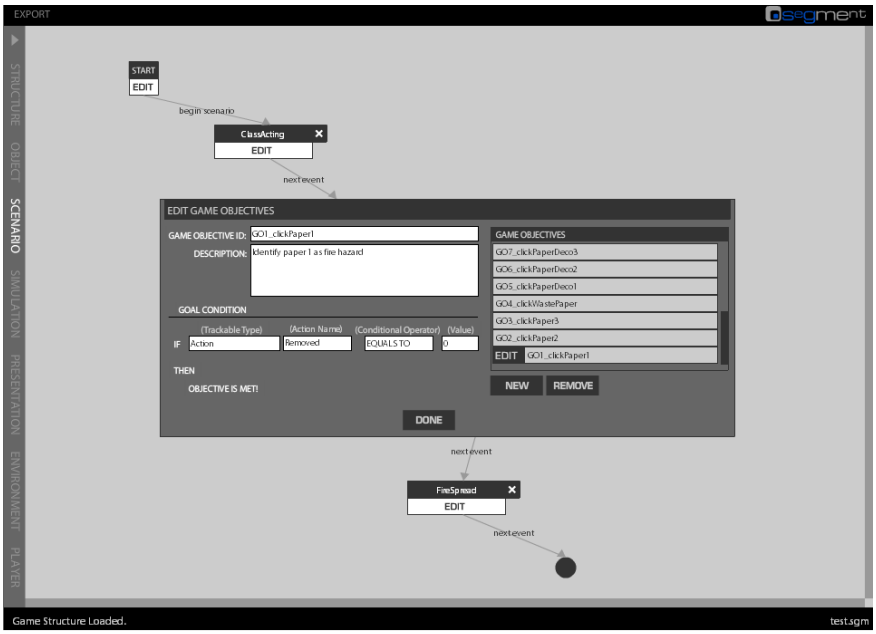

Figure 6.9: Screenshot of game objectives defined by the subject in the game scenario designer viewpoint.

The rules of this serious game dictates that the game player (via an avatar) can only interact with the other interactable game objects such as paper, waste paper basket and paper decoration. This is defined in the same viewpoint (game scenario designer) as shown in Figure 6.10.

Figure 6.10: Screenshot of game rules defined by the subject in the game scenario designer viewpoint.

Progressing from the previous stage, the subject modelled each game presentation in the serious game using game presentation designer viewpoint. The simple "Introduction Screen" was composed using text and button components available in the game presentation designer viewpoint (see Figure 6.11).

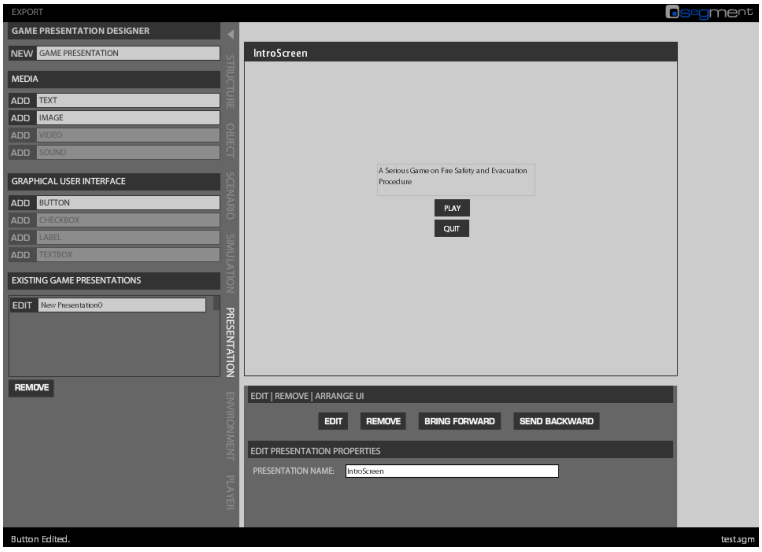

Figure 6.11: Screenshot of game presentation modelled by the subject using the game presentation designer viewpoint.

The next stage involved modelling the game simulation and this required the subject to model a user interface using the pre-set components in SeGMEnt and define the external forces (game physics) that may affect the game world. In the case of game scenario "Level 1", there is no external force required to be defined. The game simulation modelled by the subject is shown in Figure 6.12.

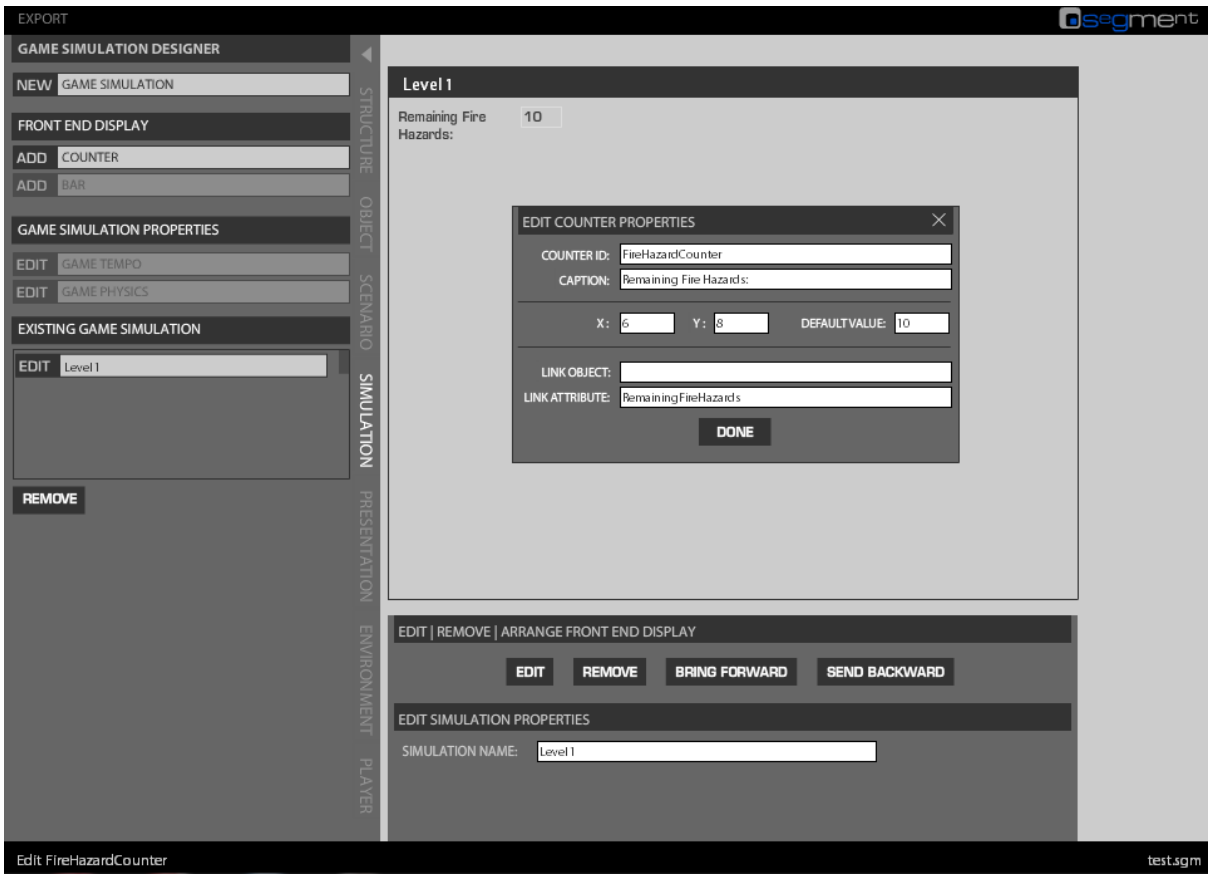

Figure 6.12: Screenshot of game simulation modelled by the subject using the game simulation designer viewpoint.

Up to this stage most of the required elements of the serious game have been modelled and this will allow designers to progress and model the flow and the structure of the game. The flow and game structure defined by the subject based on the design of the serious game as illustrated in Figure 6.3 and the result is shown in Figure 6.13.

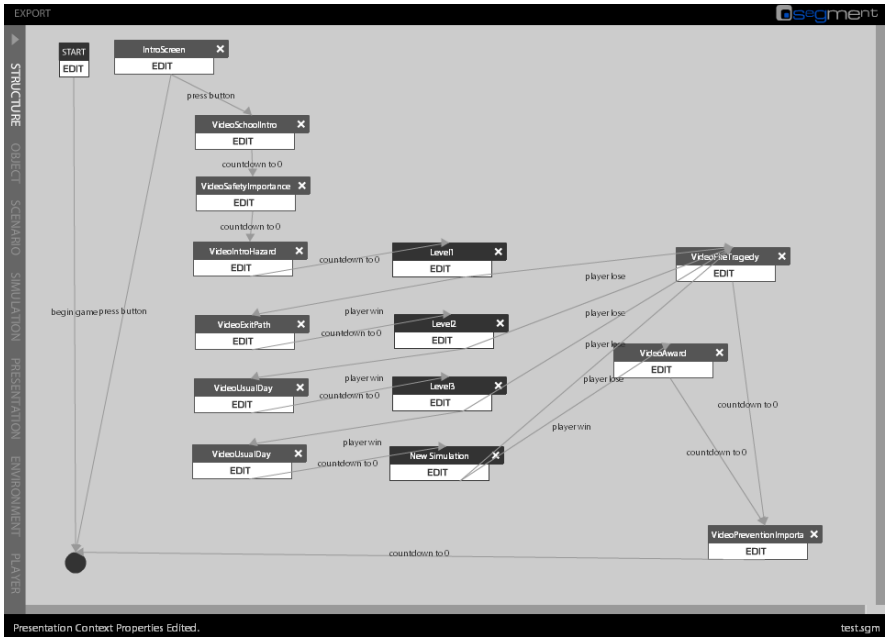

Figure 6.13: Screenshot of game structure modelled by the subject using the game structure designer viewpoint.

The final stage of the modelling serious game in SeGMEnt involves the definition of game player. This requires the subject to specify the avatar for the game player, the inventory size, the game attributes associated to the game player, game record that logs the game player's performance and the control mechanism for the game player. A screenshot of the progress is shown in Figure 6.14.

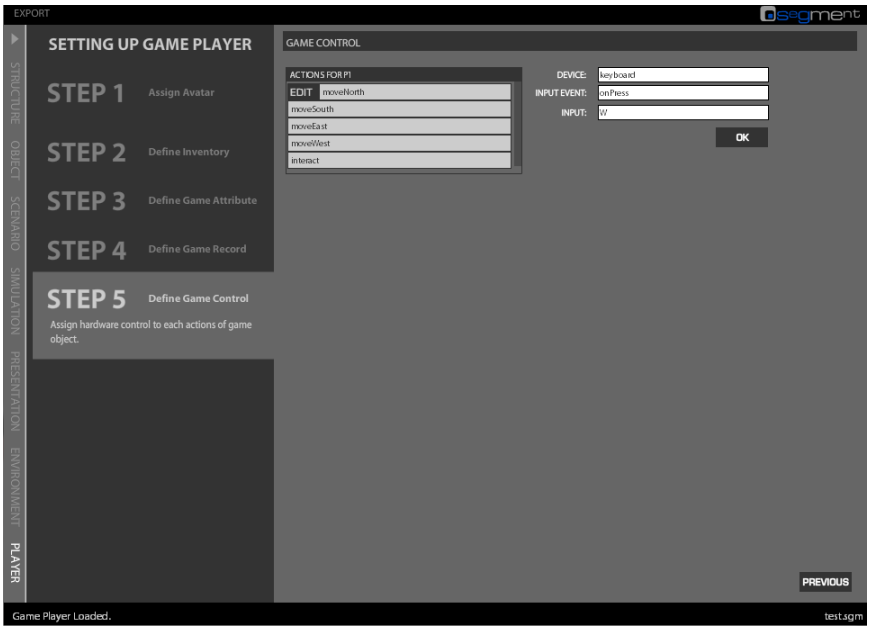

Figure 6.14: Screenshot of game control defined by the subject using the game player designer viewpoint.

6.1.3 Automated Transformation and Code Generation

The transformations of models and generation of code in our model driven framework is automatic and can be initiated with through the "Export" command in SeGMEnt. All the subject has to do was press the "Export" button in SeGMEnt. The "Export" command will first generate an XML file compliance (gameContentModel.xml) with the Game Content Model by passing the data from SeGMEnt to Game Content Model Creator (gameContentModelCreator.php). After the file has been created, the Game Content Model Creator passes the control to Game Technology Model Translator (gameTechnologyModelTranslator.php) which will read the Game Content Model (gameContentModel.xml) and transform it into a programmable format which is also in XML file format. A new XML file named gameTechnologyModel.xml is then created. Once the Game Content Model has been transformed to the Game Technology Model, control is then passed to Game Software Model translator which will read the Game Technology Model (gameTechnologyModel.xml) and add in platform specific information to the Game Technology Model to form the Game Software Model (gameSoftwareModel.xml). Once transformation is complete, the control is then passed to the ActionScript Generator which generates ActionScript 2.0 code in the form of text output presented in a web interface. In our prototype, it requires the code to be copied into the Adobe Flash CS3 Integrated Development Environment (IDE). However, this can be further developed to allow game artefacts to be automatically generated using framework such as MING PHP (http://php.net/manual/en/book.ming.php), an open source library for creation of SWF files (Flash files). In our case study, the generated code was copied into the Adobe Flash CS3 IDE by us to demonstrate the serious game application generated from our tools as shown in Figure 6.15.

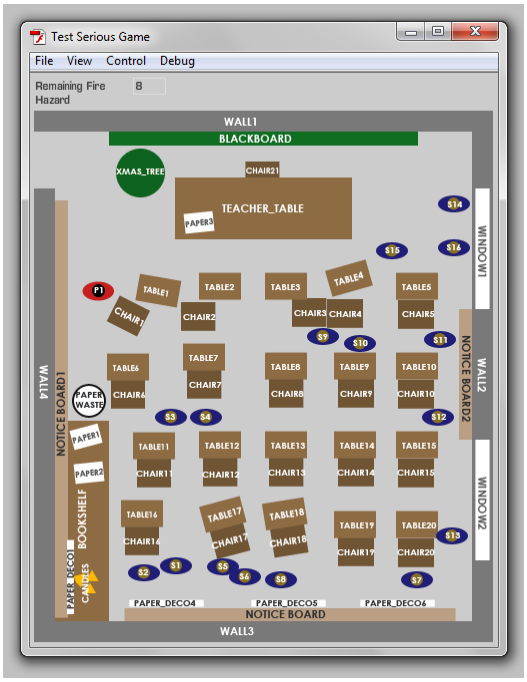

Figure 6.15: Screenshot of game prototype generated from our model driven software development framework tested on Adobe Flash platform.

6.1.4 Findings and Analysis of Observation

One of the findings from our observation throughout the case study is the learning curve required when designing serious games. We found that without our guidance, designing serious games can be a challenging task on its own for a non-technical domain expert. This is due lack of knowledge, understanding and experience in game design. It became clearer for the subject once we have showed her how to design the serious game from a lesson design point of view. In comparison to traditional game design, we found that lesson design does not differ much except that game design is a meticulous process.

We also found that although SeGMEnt was designed with assistive user interfaces and to encapsulate the technicality of serious games development, it still requires non-technical domain experts to familiarise themselves with such a tool. There are moments during the case study where subject needs to seek advice and clarification on the use of the modelling tool. This is inevitable and common with any application or software package. However, the learning curve can be made less steep with the availability of learning resources and we expect user to become more familiar with the modelling environment over time with more practice.

Another aspect which was highlighted in the case study is the methodical approach of modelling that users have to adhere to. Modelling is bounded by rules that govern the structure and relationship of the data defined by the concepts in a serious game. The systematic approach can be perceived as a

restriction by users but it is necessary to ensure that the Game Content Model produced by users using our SeGMEnt tool is valid.

One of the measurable factors in our case study is the amount of time spent in producing the serious game. In our case study, the subject spent approximately five hours with our guidance to model the serious game (all the levels) using SeGMEnt and the transformation of models and generation of code takes almost no time. This excludes the time spent designing the serious game which was described in Section 6.1.1. We expect the time spent in modelling may increase for someone who does not have computing background or above moderate IT literacy skill. In comparison, this would take approximately 20 man-days of someone with at least 4 year of formal training in computer games development to produce such codes based our professional estimate. This is a significant improvement of productivity.

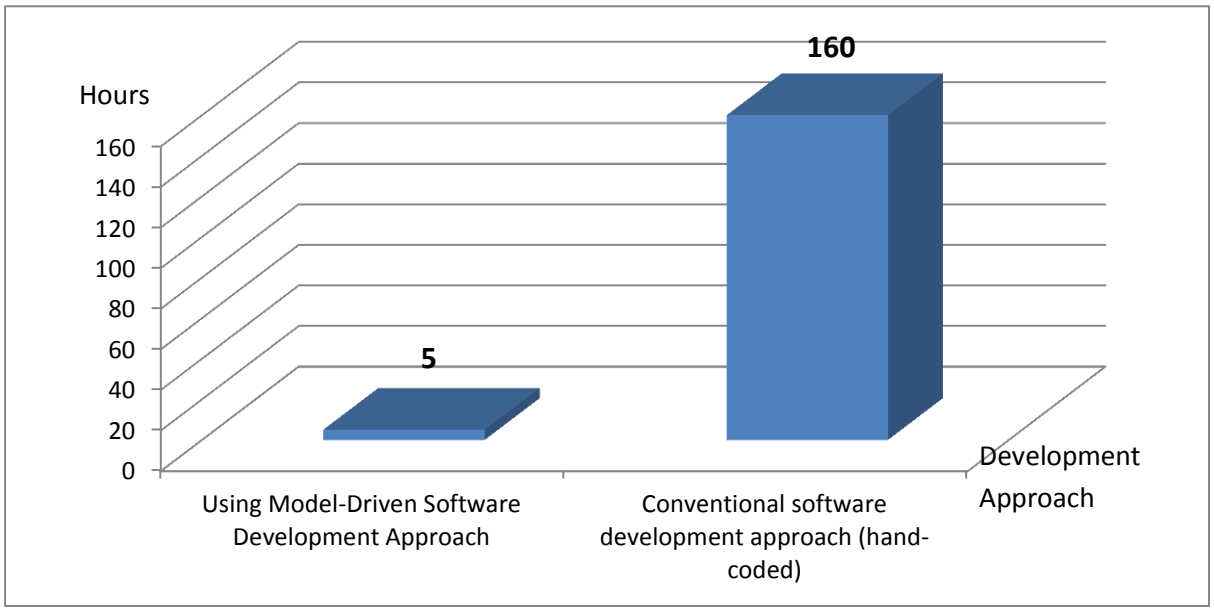

Figure 6.16: Comparison of estimated hours required to produce code for the fire safety and evacuation procedure serious game

In terms of costing, it would cost approximately £7,000.00 based on the rate of £350 per-man-day (rates vary with experience, location and project) for an experienced freelance Adobe Flash Developer to produce the code (the cost is higher if project is outsourced to a studio). In fact this would cost more as serious games tend to follow an iterative cycle of design revision to ensure the game produced is intended primarily for learning. Using our model-driven approach, non-technical domain experts are freed from such financial commitment and it lowers the barrier to adopt game-based learning.

It is a known fact that tools can simplify a mundane or complex task. In our case, we know that the collection of tools we produced based on our model-driven software development framework can help to reduce complexity of serious games development for non-technical domain experts. During the case study, we did ask the subject if she could produce such software code based on the serious game design she drafted. The immediate reply was "I have no idea how to begin produce this in software code because I have no experience with programming games. It would take me maybe a long time to produce something like this". This is evident that the collection of tools in our model-driven framework is serving its purpose – to reduce complexity and hide the technicality of game development from non-technical domain experts.

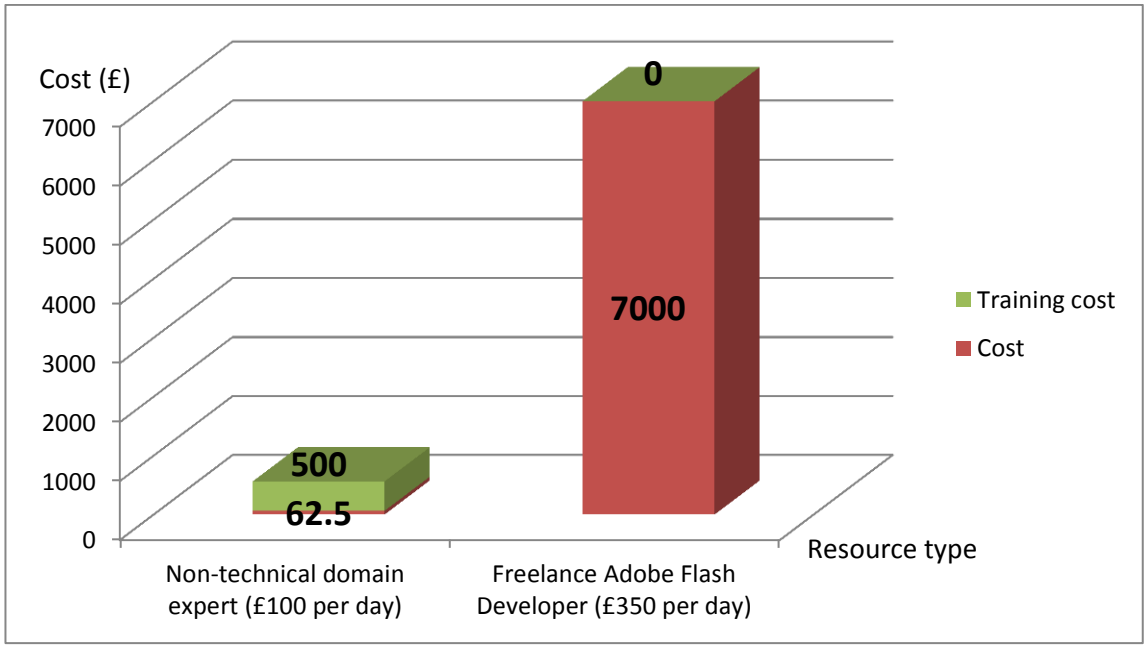

Figure 6.17: Comparison of estimated cost required to produce code for the fire safety and evacuation procedure serious game

Finally, we did ask if the subject would consider using the game-based learning approach as an alternative learning approach with the aid of tool such as SeGMEnt, model translator and model generator. The reply was moderate to highly probable instead of a convincing "yes". We believe this is because serious games design and development is a specialised area which demands specialised skillsets. Nevertheless this is still a very positive feedback and it proves that availability of tool such as SeGMEnt, model translators and code generator can lower the barrier towards the adoption of game-based learning.

6.2 Evaluation

In the following sections, we present evaluations of our model-driven framework, modelling environment, MDE tools and case study conducted.

6.2.1 Evaluation of the new Framework

Our model-driven serious games development framework presented in Chapter 5 uses a three-tier model to model a variation of serious games derived from concepts used in designing simulation games and role-playing games.

As described in Chapter 5, the Game Content Model is designed to represent serious game design formally whereas the Game Technology Model represents the game programmatically. This separates the game content and logic from the technology. The benefit of such separation is the flexibility to represent the same game model in different implementation and game technology. The Game Software Model as described represents the game as software specific to a technology platform. It builds on the Game Technology Model to include platform specific components which are omitted in the Game Technology Model. The advantage of such a separation is the ability to reuse the majority of the implementation of game software on various targeted platforms.

However, the three-tier model does have its limitations. A change of model will affect the tool chains. However, this is normal in the MDE process. This also happens in the real world when there is a change in software model, the implementation will need to be revised to accommodate for the changes. Likewise, in MDE the change in the model will require updating in the modelling tool to include new notation and translators and generators will need to be updated to accommodate such change. Sometimes this may involve a revision of the MDE tools due to the complexity introduced through the change. Therefore, it is crucial that model is well-defined to cover the range of software variation required to generate to the targeted platforms.

The Game Content Model proposed in Section 5.3 was designed specifically to aid the modelling of simulation and role-playing game genre of serious game for game-based learning. As we have explained earlier, the simulation and role-playing genres have many commonalities with other game genres and by disassembling the concepts from both genres into individual building blocks it gives domain experts the tools to formally specify the design of their serious game. Although this has sufficient concepts to cover a variety of serious games which domain experts could produce serious, we fell that creative and innovative use concepts from other game genres may also yield positive

learning experience when it is designed with pedagogy in mind. Therefore it would be interesting to extend our Game Content Model to support concepts from other game genres in the future.

Another aspect of Game Content Model which needs to be evaluated is the support for design of virtual learning activities. As we have described in Section 2.1, learners can learn through game-playing and studying the properties and behaviour of in-game components, the relationship between these in-game components and the solving of problems in the defined scenario. In the Game Content Model, domain experts can specify the properties and behaviour of in-game components using the Game Object concept presented in Section 5.3.4. The relationships between the in-game components can be defined using Game Interaction Rule presented in Section 5.3.8.1 whereas relationship with the environment can be governed through the definition of Game Physics in Section 5.3.3.3. For problem solving activities, domain experts can design a virtual situation using the Game Scenario as described in Section 5.3.5 to test learners' understanding and application of knowledge in the designed situation. From the context of lesson design, the Game Structure presented in Section 5.3.1 provides domain experts the building blocks to plan the flow of the lesson in accordance with Gagne's Nine Instructional Events (Gagne, 1970). The difficulty indicator described in Section 5.3.5 also provides a means for domain experts to organise lessons (or game scenarios) in an increasing difficulty fashion. Overall, the Game Content Model does offer the necessary support for domain experts to design meaningful play. However, there are certain aspects of teaching and learning that are not within the scope of the present Game Content Model such as monitoring and reporting of learner's performance, support for teaching of kinaesthetic and motion, and the inclusion of game design patterns to aid domain experts to meaningful-play-activity. These will be proposed as further work for this research.

The game specific system in Game Technology Model proposed in Section 5.4 only supports simulation and role-playing genre. The generation of artefacts in the case study showed that Game Technology Model is a valid implementation of game software. However it does not indicate if the implementation is optimal. The measurement of effectiveness and performance of artefacts generated is not within the scope of this research study.

6.2.2 Evaluation of SeGMEnt (Serious Games Modelling Environment)

The prototype we implemented to demonstrate the applicability of our proposed framework in Appendix C is a basic modelling environment that embeds the concepts of the Game Content Model in the user interfaces and hides the technical aspects of serious game development from non-technical domain experts. In terms of user interface, different interaction models are used in different

viewpoints that best aid domain experts to model specific aspects of serious game. The state-diagram-like notations are used in designing the game structure and game scenario. The diagrams provide the necessary visualisation of the flow of the game and game-play. Additional information of the context and event are presented in dialogue window. For aspects of serious games which require positional information, we have implemented a 2D modelling environment which allow domain expert to model the game environment, presentation and scenario. An experimental step-by-step wizard was implemented for the definition of game objects and the game player. The variation of interaction models used in the SeGMEnt is unavoidable as each of the viewpoints required a different form of visualisation to present the information to the users. This may requires domain experts to take time to familiarise themselves with the modelling environment and we expect they will become familiar with the SeGMEnt tool after repeated use. Each of the viewpoints is designed to help domain experts focus in modelling specific key concepts in the Game Content Model. However, there is shared information in the individual viewpoints which requires domain experts to remember and use later during the modelling of a serious game. All the UIs components developed in our prototype (See Appendix C) are designed to aid domain experts and simplify the modelling process. Whenever possible, data are fetched from other viewpoints to populate the appropriate list box for selection. The use of graphical notation in modelling the game environment is an interesting approach. However, it does not mimic the setup of a game environment at runtime. Nevertheless, it is still a useful interface and visualisation medium for domain experts to position objects or components.

An improved modelling environment is certainly in our proposal of future works for this research study.

The current language used to describe serious games in SeGMEnt is still very game design oriented. This will help a domain expert to relate to common game design concepts. A pedagogy-oriented vocabulary is avoided to prevent any confusion between concepts. However, such an idea could help teachers to relate better to the concepts in lesson design and making smoother transition from game design. This will be included in the list of future works in Chapter 7.

6.2.3 Evaluation of Model Representation

In terms of model representation, XML is chosen to make marking of information and locating of marked information easier, and therefore simplifying the task of model transformation. In addition, implementation of MDE tools is less complicated as well due to the support for XML given by most development platforms. In addition, XML is also supported by MDE technologies such as EMF and

Generic Modelling Environment (GME) for framework developers who wish to make use of existing MDE technologies. At present, both the Game Content Model and the Game Technology Model use two different schemas to represent the serious game. This is valid considering both models represent serious games in different views. It would be better if the schema representing the Game Content Model be improved to allow the Game Technology Model to be built onto. This could provide an alternative for framework developers who intend to weave additional information into the existing model to create a new model without having to completely reformat the structure of the model. This can be factored into the future work for this research.

6.2.4 Evaluation of Model Translation and Artefact Generation

One of the benefits of MDE is the translation of models and generation of software artefacts. Framework developers will need to have a deep understanding of the Game Content Model and Game Technology Model before they can transform the Game Content Model to the Game Technology Model and from Game Technology Model to Game Software Model. In our framework, we chose not to be constrained by the structure of the Game Content Model and have opted to implement our MDE tool that locates the marked information in the Game Content Model and rebuilds the Game Technology Model and Game Software Model from scratch. Development of the MDE transformation tool is proven to be much simpler and straight forward especially with modern XML programming interfaces such as Simple XML in PHP 5.0.

The evaluation of the performance of our model translation and artefacts is not within the scope of this research study. However, it is a known fact that generated code can lose out on code optimisation. This would limit the complexity level of the serious game and the number of dynamic objects the software can process at runtime. Unlike generated code, manual hand-coding permits game developers with advanced knowledge to apply clever solutions to unblock performance bottle-neck manually. This is a trade-off between conventional approach of games development and the model-driven approach. From the domain expert's point of view it is the least of their worries.

6.3 Chapter Summary

In this chapter, we have presented a case study to demonstrate the applicability of our tools implemented based on our model-driven serious game development framework presented in Chapter 5 and evaluated this research study from various perspectives.

In our case study (see Section 6.1.4), we presented the planning phase and the prototyping phase of serious game design lifecycle to uncover the issues related to serious games production particularly on the use of our tools. Our finding from the case study affirms that this approach can help non-technical domain experts in production of serious games and it also increases the likelihood of adopting game-based learning as an alternative teaching and learning approach. However, we found that serious games design is still a creative process and demands specialised skill despite the tools being a guide and aid for non-technical domain experts. This result is expected as we cannot expect our tools to instantly turn a novice serious game designer to a serious game designer who is capable of designing interesting and creative problems for the game players.

In our evaluations of this research project, we have critically evaluated the framework, the SeGMEnt tool, the choice of our model representation, the model translation tools and code generation tool, and finally the case study we conducted. We also acknowledged that the tools we developed for this research project and the case study conducted are not without limitations. However, this does not distract from the intended aims of this research study. We have developed a model-driven framework to support the development of serious games and proven its application through the tools we developed. We expect that with the right expertise and resources, better and more powerful tools can be developed using our model-driven serious games development framework. We will discuss this in greater depth in Chapter 7.

CHAPTER 7 - CONCLUSIONS AND FUTURE WORK

This chapter concludes the work in this thesis. In the following sections, we present a summary of the work done in this research study and the outcome of this study. Contributions made from this research study are highlighted before we present our thoughts and views to further our works.

7.1 Conclusions

Game-based learning is a highly desired technology-assisted learning approach for the "PlayStation-driven" generation of learners. However, it lacks technological solutions that can help non-technical domain experts to author custom interactive learning content. This is the initial motivation that influences our research study.

In this research study we proposed a novel model-driven framework that supports the development of games (Chapter 5) and applying it to the domain of serious games for game-based learning (Appendix C). By infusing game development with practices of MDE, we have implemented SeGMEnt, a high-level serious game authoring environment that helps non-technical domain experts to produce serious games quickly, easily and affordably (in the long term). The complexity of serious game development is now hidden behind the SeGMEnt and driven by the models and the MDE tools that interpret and refine the models for the generation of software artefacts.

Using our model-driven approach non-technical domain experts can model a game by providing the necessary details that are required to compose a Game Content Model using our SeGMEnt tool (Appendix C.2). The Game Content Model acts as a game design template with a collection of customisable design blocks which aid domain experts to formally design a game. The serious game design in Game Content Model format gets translated into the Game Technology Model where data are formatted into a programmable structure using model the Game Technology Model translator presenting a computational independent view of the serious game. Additional platform specific information is added to the Game Technology Model by the Game Software Model translator during the translation from the Game Technology Model to the Game Software Model to produce functional serious game software targeted at a specific technology platform.

This model-driven approach changes how computer games are developed. Instead of developing software based on a set of given design requirements, this model-driven approach demands software developers to produce assets and tools which non-technical domain experts can use to produce computer games with ease.

Our prototypical implementation (Appendix C) has demonstrated the application of the proposed model-driven game development framework to the field of game-based learning. Our SeGMEnt tool provides teachers with building blocks to design a range of serious games through a graphical UI. The MDE tools in our prototype showed that the serious game model produced using the SeGMEnt can be transformed into a platform independent model (Game Technology Model) and subsequently to a platform specific model (Game Software Model) before a software artefact is generated.

Our finding from the case study has shown that our proposed model-driven serious game development approach can help lower the barriers, both technically and financially, for non-technical domain experts to produce serious games. Using our approach non-technical domain experts can focus on the serious game design and production process without worrying about the technical aspects of development which have been encapsulated by the SeGMEnt tool. This is a positive step towards for the game-based learning communities.

In mainstream game production, the growing number of amateur and hobbyist game developers wanting to produce "indie games" for both commercial and non-commercial purposes has fuelled the advancement of high-level game authoring tools such as Game Salad[37], Unity 3D[38] and Unreal Development Kit (UDK)[39]. These tools provide facilities for game modelling using a GUI and support for the generation of software artefacts for a range of popular operating platform (especially for the mobile platforms Apple iOS, Google Android and Microsoft Windows Phone 8). These technologies are examples of the trends in high-level user specific tools and some of these tools (such as Game Salad) are suitable for non-technical domain experts. We envision that such trends will continue and our model-driven game development framework will serve as a basis for more implementation of high-level game authoring tools designed specifically for non-technical domain experts who wish to produce computer games.

In conclusion, this research was initially set out to address the two key challenges that inhibit the adoption of game-based learning described in Section 1.1 specifically (1) facilitation of serious game

[37] http://gamesalad.com/
[38] http://unity3d.com/
[39] http://www.unrealengine.com/udk/

production, and, (2) development of high-level serious development environment for practitioner of game-based learning. These two key challenges have been addressed in this research study. To address key challenge 1, we have developed a prototype of SeGMEnt (Serious Game Modelling Environment) which we described in Appendix C to aid non-technical domain expert to create serious game and the result from the findings case study validated it. For key challenge 2, we have developed a model-driven game development framework that supports the development of game development environment which we described in Chapter 5.

7.2 Contributions

This multi-disciplinary research work has made a number of novel contributions worthy of noting and many of these works have been shared with the research community in the form of publications (see List of Publications for a full list of publications from this thesis). One of the key contributions to this research is our novel model-driven serious games development framework in Chapter 5. This approach combines the knowledge from the areas of game design, game development, pedagogy and MDE to form an innovative solution which as a key reference developing a high-level games authoring environment that can encapsulate complexity of game development and automate the generation of software artefacts for the game design across a range of technology platform and operating platform.

Our novel Game Content Model is another key contribution made in this research. It is the formal and most comprehensive game design model to date. We have combined the best of GOP (Zagal, et al., 2005), RAM (Järvinen, 2007) and NESI (Sarinho & Apolinário, 2008) with our study of game design, game development and serious games to form a comprehensive, reusable, and formalised model to represent games design. Our novel Game Content Model can also be used to help novice game designers or non-technical domain experts who wish to design the computer games document design specification of a game formally. In addition, it can also be used as a tool to study the anatomy of a game both from the design and software perspectives. Our novel Game Content Model can also be extended to include specific concepts that describe components of other games genres such as action, strategy, and even puzzle. This provides the flexibility for model developers to extend our work to suit a particular purpose. In addition to the Game Content Model defined in this thesis in Section 5.3, an OWL (Web Ontology Language) ontology of the Game Content Model was produced (available to download at http://www.staff.ljmu.ac.uk/cmpotang/gamecontentmodel.html) and this is a secondary contribution to this research work.

Finally, our last contribution made from this research study is the Game Technology Model, a computational representation of serious games software independent of implementation platform and hardware, is another contribution made from this research study. It serves as a higher-level computational representation of game technology and permits functionalities of an existing game software framework to be mapped to the Game Technology Model and used in our model-driven framework. Besides its intended purpose, it can also be used as a framework for novice developers to model their own game software framework for use in game development.

7.3 Limitations

The research work presented in this thesis has some limitations due to time and resource constraints. The scope of our modelling tool (SeGMEnt) has been limited to only the modelling of 2D-based serious games of simulation and role playing genres due to the amount of development work required to demonstrate the model-driven approach. This will present some constraints to the way serious game can be designed and in some way can result to a less complex game. However, this can be scaled up to incorporate the expanded concepts described in Game Content Model in order to accommodate more complex serious game.

At present, model translators are hand-coded and are only able to translate the tokens supported in the model. Additional tokens generated from the modelling environment will require further development and this is part of practice in model-driven approach. It is unlikely to have new tokens added to existing model as the interdependencies of the token may result in the re-engineering of the platform independent model. This is also the case for the generation of codes which is also hand-coded and is only able to generate codes of a specific platform.

The supported target technology platform is limited in this study. This is due to the large amount of work needed in order to understand the targeted technology platform and to develop the necessary code generator for a targeted technology platform making the support of more technology infeasible in this research study. Nevertheless, the prototypical implementation has proven that it is possible and given more resource it is feasible to support more technology platforms.

Finally, the current modelling environment lacks visualisations to enable domain experts to model a game environment as it would appear in run-time. This will require further development but the present approach does not distract domain experts from providing approximate positional data of a component using the visual notations provided.

7.4 Future work

There are many areas which can be improved to further this relatively new and exciting area of research. One of the areas of improvement we propose is the modelling environment. At present, SeGMEnt only facilitates the design of presentation, simulation and game environment using visual notation to provide approximate representation of the "look and feel" of the visual designs. This can be further developed to provide a more accurate "What You See Is What You Get" design or modelling environment similar to the interface designer provided by IDEs such as Visual Studio and level editor developed by Unity3D and Unreal. This would require an advanced visualisation implementation using 3D graphic libraries.

In addition, the modelling environment should provide support for modelling of both 2D and 3D games. In order to support production of 3D, the MDE tools need to be adjusted to include tokens of information representing 3D implementations. It would be interesting to have the MDE tools be able to pair the tokens with new code templates without many changes to the implementation. This is possible and will require abstract transformation implementation which can be a whole new research area on its own.

It would be interesting to also include a pedagogic specific workspace in the modelling environment. A pedagogic specific workspace would have the same UI environment but UI components will be labelled with pedagogic-oriented vocabulary. This could potentially help to reduce the learning curve and make game design similar to the task of lesson design in some way to teachers. Such a feature should only be implemented at the UI level rather than making changes to the Game Content Model to prevent any confusion to framework developers who are implementing the MDE tools.

Furthering the research on the modelling environment, a thorough usability evaluation can be conducted on the modelling environment to gain more insights on user's behaviour and preferences which was not in the scope of this research study. More user testing can be conducted with larger sample group. Findings from this study can then be used as input to further improve the interface and interaction design of the modelling environment. As part of the user experience design process, this will require a few test-design-prototype cycles before we can achieve a design that is ideal for non-technical expert.

The next proposal to further this research is to extend the Game Content Model to include concepts for monitoring and reporting learners' performance. At present this can be achieved through

the definition of a game record and analytics is achieved by specification of traceable data. It would be interesting to include other behavioural information such as learning patterns, learning preferences, usage behaviours and others which are useful for teachers. This can provide useful pointers to areas of the serious game which requires refinements. In addition, this information could also help teachers to plan individual student learning for those students who are experiencing difficulty in learning specific material in the serious game.

Furthering research on Game Content Model is a multi-genre support Game Content Model which can be supported by different Game Technology Models. This would allow domain experts to model serious games of different genres. At present, the Game Content Model covers most of the concepts for designing simulation and role playing games and is supported by the Game Technology Model. Development of a multi-genre support Game Content Model will require addition of genre-specific concept and more importantly the relevant Game Technology Model that can support the operation of the specific genre. An algorithm could be developed to check which type of genre the domain expert is modelling and subsequently pair it with the compatible Game Technology Model.

Another possible area to further this research includes the addition of game design patterns as described in the work of Björk, Lundgren, & Holopainen (2003) into Game Content Model to aid domain expert in creating gameplay. It defines the characteristic of a play and allows domain expert to customise the game design pattern to suit their needs. Identified game design patterns suitable for use in serious games can be defined using concepts in Game Content Model and be made available for domain expert in the modelling environment as pre-defined scenario. This will simplify the game design process and domain experts will only need to provide the necessary content to complete the scenario.

A potential area to further this research is the support of new generations of gesture based inputs to extend computer games into training of motion and kinesthetic. The game control described in Section 5.3.9.4 can be expanded to support input devices such as Microsoft Kinect, Sony PlayStation Move and Nintendo WiiMote. This can pave way for the creation of more innovative games relating to the sports, physiotherapy, and health and safety domain which require detection of physical movements or replicating a specific posture.

Another area which we can further into from this research is the mobile segment. The thought of using mobile games in learning is not new. This can be linked to the advancement of mobile devices, mobile gaming trend and the introduction of games-based learning. Designing and developing serious

mobile games introduces a set different set of challenges due to the limitations of hardware interface (or perhaps the advantage the gesture-based interaction it can offer) and the short-burst usage pattern. This will require further elaboration of the input interface in Game Content Model and perhaps a leaner version of Game Technology Model that caters for the mobile platform.

The Game Technology Model described in Section 5.4 focuses mainly on single user (player) mode. This can be extended to include online multiplayer support for collaborative game-based learning where multiple users can role-play and solve problem in the same scenario either as collaboratively as a unit or taking the other designated role in the serious game. This will require an online multiplayer component which uses the networking service component to manage aspects such as authentication of players, setting up of multiplayer game session and transmission of data packets. In order to achieve this, it will require reengineering of the Game Technology Model to account for such feature.

It would also be interesting to further this research by studying the versatility of the Game Content Model and Game Technology Model proposed in Chapter 5. This will require more case studies to be conducted. In addition, the performance of the generated artefacts could also be measured to provide insights into how well such a model copes at runtime on different technology platform.

Finally, the scope of high-level serious games authoring environment can be further expanded to support existing learning management systems such as Blackboard. The integration between the high-level serious games authoring environment and learning management system could further simplify the process for teachers to publish, manage and distribute their active learning content.

7.5 Concluding Remarks

This thesis provides a novel model-driven games development framework to aid non-technical domain expert in the development of computer games. The models are the core of this approach. This approach enables domain experts to design a computer game formally using the building blocks defined in the model in a UI-based modelling environment which hides all the technology intricacies related to game development. The formalised design of the computer game is translated into a more refined representation of game software and subsequently be transformed into a game software using MDE tools developed by framework developers.

Collectively, the model-driven framework offers a renewed software engineering approach towards game development where domain experts take the role of designers and game developers (programmers and digital content creators) provide the necessary tools and assets, and the entire

process is centred on the model. This work has made a significant impact on game-based learning research and has addressed two key challenges pertaining to game-based learning identified in Section 1.1. The evidences of this are in the list of research publications produced throughout this research study. It has also opened up a number of new research areas to further the work in this research study as described in Section 8.4.

We hope that this work will simplify the development of games and would aid non-technical domain experts to produce games for use in their respective domains. We also hope that this would take us step nearer towards the mass adoption of games-based learning and will bring forth a revolution in education technology from the passive 'electronic' learning to an active and 'effective' electronic learning experience that addresses the learning styles of the 21st century learners. And we believe that the proposed future works can bring about greater impact on the research community and society.

APPENDICES

APPENDIX A: Ontology for Game Content Model

Table A.1: Ontology for Serious Game & Game Structure in BNF Representation

Concept	BNF Representation
Serious Game	<SeriousGame> ::= <Title> <Author> <GameStructure> <GamePlayer>
Game Structure	<GameStructure> ::= <GameStructureType> <GameContext> {<GameContext>}
Game Context	<GameContext> ::= [<GamePresentation> | <GameStructure>] <EventTrigger> {<EventTrigger>} <PedagogicEvents>
Pedagogic Event	<PedagogicEvents> ::= <EventOfInstruction> {<EventOfInstruction>}
Event Trigger	<EventTrigger> ::= <Identifier> (<InputTrigger> | <TimeTrigger> | <ProximityTrigger> | <GameMechanicsTrigger>) <TriggerTarget>
Input Trigger	<InputTrigger> ::= <InputInterface>
Time Trigger	<TimeTrigger> ::= <TimeInterval> <Repeat>
Proximity Trigger	<ProximityTrigger> ::= <Position> <Area>
Game Mechanics Trigger	<GameMechanicsTrigger> ::= <GameApplicationEvent>
Input Interface	<InputInterface> ::= (<HardwareInterface> | <GUIInterface>)
Hardware Interface	<HardwareInterface> ::= <HardwareType> <InputEvent>
GUI Interface	<GUIInterface> ::= <ReferenceToGUI>
Position	<Position> ::= <2DPosition> | <3DPosition>
Area	<Area> ::= <2DArea> | <3DArea>

Table A.2: Ontology for Game Presentation in BNF Representation

Concept	BNF Representation
Game Presentation	<GamePresentation> ::= <Identifier> <MediaComponent> {<MediaComponent>}[<GUIComponent> { <GUIComponent>}] <DepthIndex> <Dimension> <2DPosition>
Media Component	<MediaComponent> ::= <MediaText> | <MediaImage> | <MediaVideo> | <MediaSound>
Media Text	<MediaText> ::= <Identifier> <Caption> <2DPosition> <2DDimension> [<TextFormatting>]
Media Image	<MediaImage> ::= <Identifier> <Source> <2DPosition> <2DDimension>
Media Sound	<MediaSound> ::= <Identifier> <Repeat> <Source>
Media Video	<MediaVideo> ::= <Identifier> <Source> <2DPosition> <2DDimension>
Dimension	<Dimension> ::= <2DDimension> | <3DDimension>
GUI Component	<GUIComponent> ::= <GUIButton> | <GUIListbox> | <GUIRadiobutton> | <GUITextbox>
Button	<GUIButton> ::= ID> <2DPosition> <2DDimension> <GUIStyleID> <backgroundImage> <Caption>
Check Box	<GUICheckbox> :: <Identifier> <2DPosition> <Caption> <Value> <GUIStyleID>
List Box	<GUIListbox> ::= <Identifier> <2DPosition> <Width> <Caption> <listValue> { <ListValue>} <GUIStyleID>
Radio Button	<GUIRadiobutton> ::= <Identifier> <GroupID> <Caption> <Value> <2DPosition> <GUIStyleID>
Text Box	<GUITextbox> ::= <Identifier> <2DPosition> <2DDimension> <Caption> <GUIStyleID>

Table A.3: Ontology for Game Simulation in BNF Representation

Concept	BNF Representation
Game Simulation	<GameSimulation> ::= <Identifier> <GameDimension> <GameTempo> <GamePhysics> <FrontEndDisplay> {<FrontEndDisplay>} <GameScenario>
Game Tempo	<GameTempo> ::= <RealTime> <VirtualTime>
Game Physics	<GamePhysics> ::= <CollisionWorld> <EnvironmentalForce> { <EnvironmentalForce> }
Environmental Force	<EnvironmentalForce> ::= <force>
Force	<Force> ::= <Identifier> <ForceValue> <ForceAngle>
Force Angle	<ForceAngle> ::= <StaticForceAngle> | <DynamicForgeAngle>
Static Force Angle	<StaticForceAngle> ::= <Angle>
Dynamic Force Angle	<DynamicForceAngle> ::= <MinAngle> <MaxAngle>
Min Angle	<MinAngle> ::= <Angle>
MaxAngle	<MaxAngle> ::= <Angle>
Force Value	<ForceValue> ::= <StaticForceValue> | <DynamicForceValue>
Static Force Value	<StaticForceValue> ::= <ForceValue>
Dynamic Force Value	<dynamicForceValue> ::= <MinForceValue> <MaxForceValue>
Min Force Value	<MinForceValue> ::= <ForceValue>
Max Force Value	<MaxForceValue> ::= <ForceValue>
Front End Display	<FrontEndDisplay> ::= <Position> <FrontEndDisplayType> <FrontEndDisplayStyle> (<StaticFrontEndDisplay> | <DynamicFrontEndDisplay>)
Static Front End Display	<StaticFrontEndDisplay> ::= <DataSource>
Dynamic Front End Display	<DynamicFrontEndDisplay> ::= <Owner> <DataSource> <RelativePositioning>

Table A.4: Ontology for Game Objects in BNF Representation

Concept	BNF Representation
Game Object	<GameObject> ::= <Identifier> <Position> <GameObjectType> <GameTheme>
Game Object Type	<GameObjectType> ::= <ActorObject> | <EnhancementObject> | <ConsumableObject> | <ItemObject> | <MechanicalObject> | <ProjectileObject> | <StructuralObject> | <DecorativeObject> | <SurfaceObject>
Actor Object	<ActorObject> ::= <ObjectAttributes> <ObjectAppearance> <ObjectAction> {<objectAction>} <objectIntelligence>
Enhancement Object	<EnhancementObject> ::= <ObjectAttributes> <ObjectAppearance> <ObjectAction> {<objectAction>}
Consumable Object	<ConsumableObject> ::= <ObjectAttributes> <ObjectAppearance> <ObjectAction> {<objectAction>}
Item Object	<ItemObject> ::= <ObjectAttributes> <ObjectAppearance> <ObjectActions>
Mechanical Object	<MechanicalObject> ::= <ObjectAttributes> <ObjectAppearance> <ObjectAction> {<ObjectAction>}
Projectile Object	<ProjectileObject> ::= <ObjectAttributes> <ObjectAppearance> <ObjectAction> {<ObjectAction>}
Structural Object	<StructuralObject> ::= <ObjectAttributes> <ObjectAppearance>
Decorative Object	<DecorativeObject> ::= <ObjectAttributes> <ObjectAppearance>
Surface Object	<SurfaceObject> ::= <ObjectAttributes> <ObjectAppearance>
Object Attributes	<ObjectAttributes> ::= <VitalAttribute> {<VitalAttribute>} <PositionAttribute> <SolidityStateAttribute> [<MassAttribute>] [<InventoryAttribute>]

Object Appearance	<ObjectAppearance> ::= <Identifier> <ObjectImage> <ObjectInternalProjection>
Object Image	<ObjectImage> ::= <objectImageComponent> {<ObjectImageComponent>} <ObjectImageStyle>
Object Image Component	<ObjectImageComponent> ::= <Identifier> <AssetSource>
Object Internal Projection	<ObjectInternalProjection> ::= <ReferenceToFrontEndDisplay> <ReferenceToObjectAttribute>
Object Actions	<ObjectActions> ::= <ObjectAction> {<ObjectAction>}
Object Action	<ObjectAction> ::= <Identifier> <ObjectAnimation> [<ObjectMotion>] [<ObjectSpeech>] [<ObjectVitalUpdate>]
Object Animation	<ObjectAnimation> ::= <Identifier> <AnimationStartFrame> <AnimationEndFrame> <AnimationAssetSource>
Object Motion	<ObjectMotion> ::= <Identifier> <Force> <EnvironmentalForce> <SyncWithAnimation>
Object Sound	<ObjectSound> ::= <Identifier> <SoundAssetSource> [<NarrativeSource>]
Object Vital Update	<ObjectVitalUpdate> ::= <ReferenceToObjectAttribute> <ArithmeticOperator> <Constant>
Object Intelligence	<ObjectIntelligence> ::= <ObjectDecision>{<ObjectDecision>} [<ObjectAbility>]
Object Decision	<ObjectDecision> ::= <Identifier> <DecisionCondition> <ReferenceToObjectAction>
Decision Condition	<DecisionCondition> ::= <ReferenceToObjectAttribute> <ConditionalOperator> <Constant>
Object Ability	<objectAbility> ::= [<AbilityLearning>] [<AbilityNavigation>]

Table A.5: Ontology for Game Scenario in BNF Representation

Concept	BNF Representation
Game Scenario	<GameScenario> ::= <Identifier> <GameEnvironment> <VirtualCamera> {<VirtualCamera>} <DifficultyIndicator> <GameEvent> { <GameEvent>} <GameObjective> { <GameObjective>} <GameRule> { <GameRule>}
Game Environment	<GameEnvironment> ::= <Identifier> (<GameObjectID> {<GameObjectID>}) (<Checkpoints> {<Checkpoints>}) (<ProximityTrigger> { <ProximityTrigger>}) (<Light> {<Light>})
Checkpoint	<checkpoint> ::= <Position> <GameObjective>
Light	<Light> ::= <Identifier> <Position> <LightColour> <LightIntensity> (<DirectionalLight> | <SpotLight> | <PointLight> | <AreaLight>)
Directional Light	<DirectionalLight> ::= <Angle>
Spot Light	<SpotLight> ::= <Angle> <Range> <SpotAngle>
Point Light	<PointLIght> ::= <Range>
Area Light	<AreaLight> ::= <2DArea>
Virtual Camera	<VirtualCamera> ::= <StaticVirtualCamera> | <AttachedVirtualCamera>
Static Virtual Camera	<StaticVirtualCamera> ::= <Identifier> <Position> <VirtualCameraFocus>
Attached Virtual Camera	<AttachedVirtualCamera> ::= <Identifier> <AttachmentPoint> <VirtualCameraFocus>
Virtual Camera Focus	<VirtualCameraFocus> ::= (<FocusAngle> | <FocusObject>)

Table A.6: Ontology for Game Event in BNF Representation

Concept	BNF Representation
Game Event	<GameEvent> ::= <Identifier> <GameAct> {<GameAct>} [(<GameMechanicTrigger> | <TimeTrigger>)]
Game Act	<GameAct> ::= <Identifier> <GameObject> <GameActingScript>
Game Acting Script	<GameActingScript> ::= <Identifier> <GameActingCoordination> { <GameActingCoordination>}
Game Acting Coordination	<GameActingCoordination> ::= (<SimpleActingCoordination> | <ComplexActingCoodination>)
Simple Acting Coordination	<SimpleActingCoordination> ::= <Animate> | <AnimateWithGameObject> | <PlaySound> | <TranslateToCheckpoint>
Complex Acting Coordination	<complexActingCoordination> ::= (<Animate> | <AnimateWithGameObject>) [<PlaySound>] [<TranslateToCheckpoint>]

Table A.7: Ontology for Game Objective in BNF Representation

Concept	BNF Representation
Game Objective	<GameObjective> ::= <Identifier> <Description> <GoalCondition>
Goal Condition	<GoalCondition> ::= <TrackableData> <ConditionalOperator> <Constant>
Trackable Data	<TrackableData> ::= <TrackableActionCounter> | <TrackableGameAttribute> | <TrackableInputTypeCounter> | <GameScenarioCompletionTime>

Table A.8: Ontology for Game Rule in BNF Representation

Concept	BNF Representation
Game Rule	<GameRule> ::= <Identifier> (<GameScoringRule> | <GameInteractionRule>) <Description>
Game Interaction Rule	<GameInteractionRule> ::= <InterationActor> <InteractionSubject> <InteractionCondition> <InteractionOutcome>
Interaction Actor	<InteractionActor> ::= <ReferenceToGameObject> {<ReferenceToGameObject>}) | <GameObjectClass>
Interaction Subject	<InteractionSubject> ::= <ReferenceToGameObject> {<referenceToGameObject>}) | <GameObjectClass>
Interaction Condition	<InteractionCondition> ::= (<QuerySubjectOwnership> | <QuerySubjectAttribute> | <QueryGameAttribute>)
Interaction Outcome	<InteractionOutcome> ::= (<updateGameAttribute> | <updateGameObjectAttribute>)
Game Scoring Rule	<GameScoringRule> ::= <ScoringCondition> <ScoringOutcome>
Scoring Condition	<ScoringCondition> ::= <QueryGameObjectState> | <QueryInput> | <QueryDuration> | <QueryOtherGameObjectives>

Table A.9: Ontology for Game Player in BNF Representation

Concept	BNF Representation
Game Player	<GamePlayer> ::= <GamePlayerID> <Avatar> <GameAttribute> {<GameAttribute>} <Inventory> <GameControl> {<GameControl>} <GameRecord> {<GameRecord>}
Avatar	<Avatar> ::= <StaticAvatar> | <DynamicAvatar>
Static Avatar	<StaticAvatar> ::= <ReferenceToGameObject>
Dynamic Avatar	<DynamicAvatar> := <ReferenceToGameObject> {<ReferenceToGameObject>}
Inventory	<Inventory> ::= <InventoryAttribute>
Game Attribute	<GameAttribute> ::= <Identifier> (<GameAttributeVital> | <GameAttributeScore>)
Game Attribute Vital	<gameAttributeVital> ::= (<ReferencedVitalAttribute> | <UnreferencedVitalAttribute>)
Game Control	<GameControl> ::= <Identifier> <InputInterface> <ActionMap>
Action Map	<ActionMap> ::= <ActiveObjectState> <ReferenceToObjectAction>
Game Record	<GameRecord> ::= <Identifier> <GameResult>
Game Result	<GameResult> ::= <RawGameResult> | <ComputedGameResult>

Table A.10: Ontology for Game Theme in BNF Representation

Concept	BNF Representation
Game Theme	<GameTheme> ::= <Description>

APPENDIX B: Game Technology Model

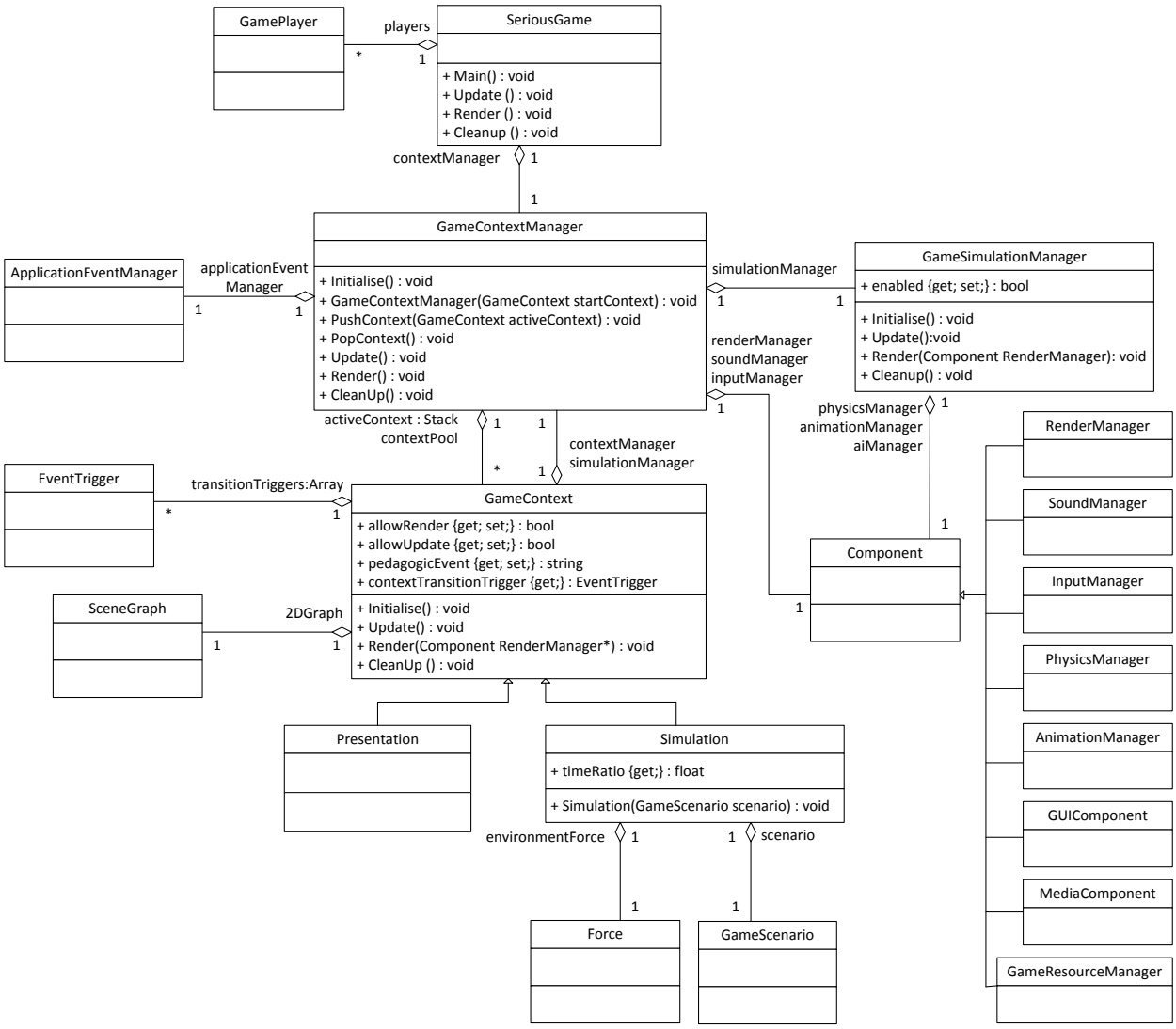

Figure B.1: UML Diagram for Game Context System (Game Context Manager) and Game Simulation System (Game Simulation Manager)

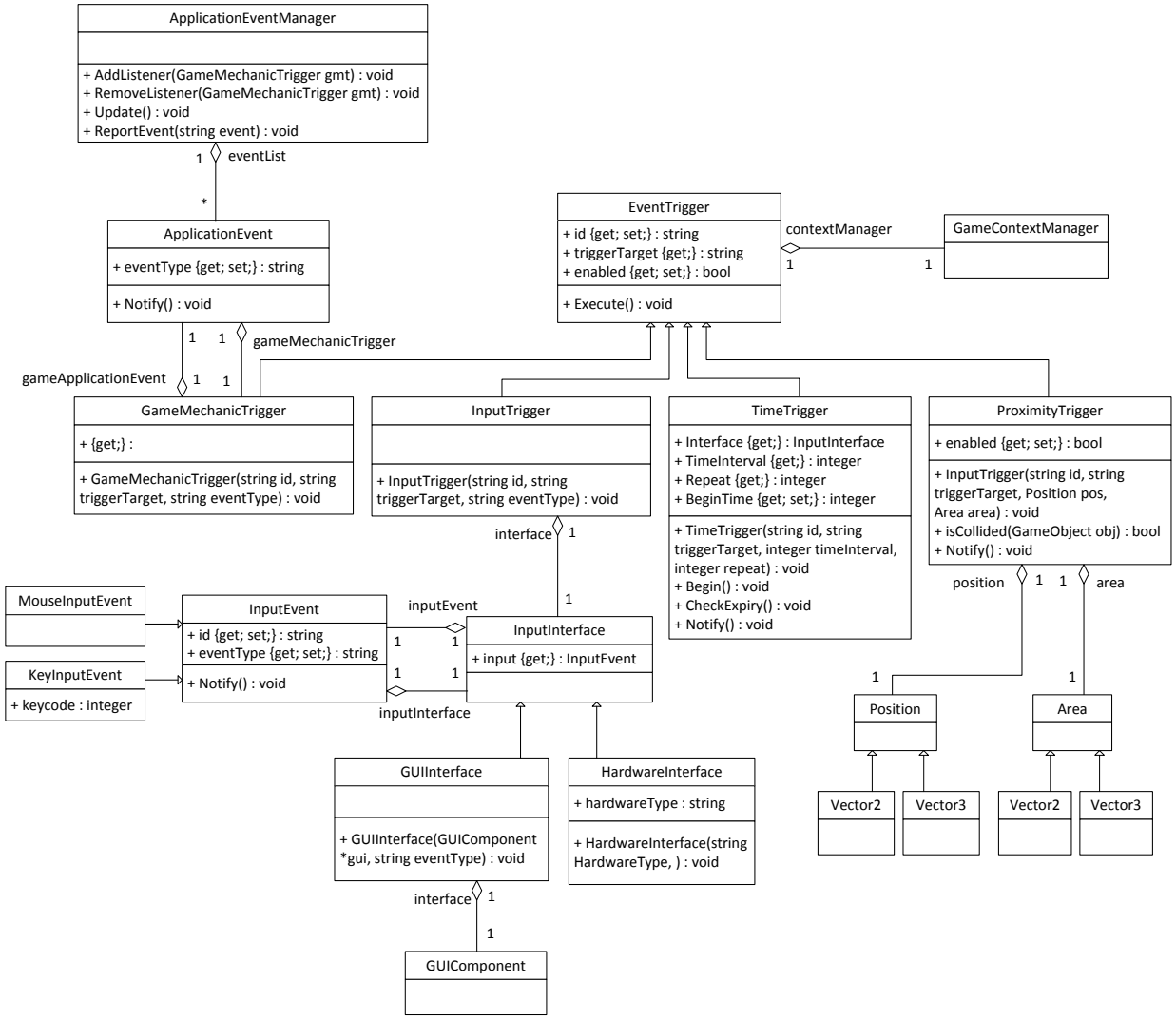

Figure B.2: UML Diagram for Event Trigger

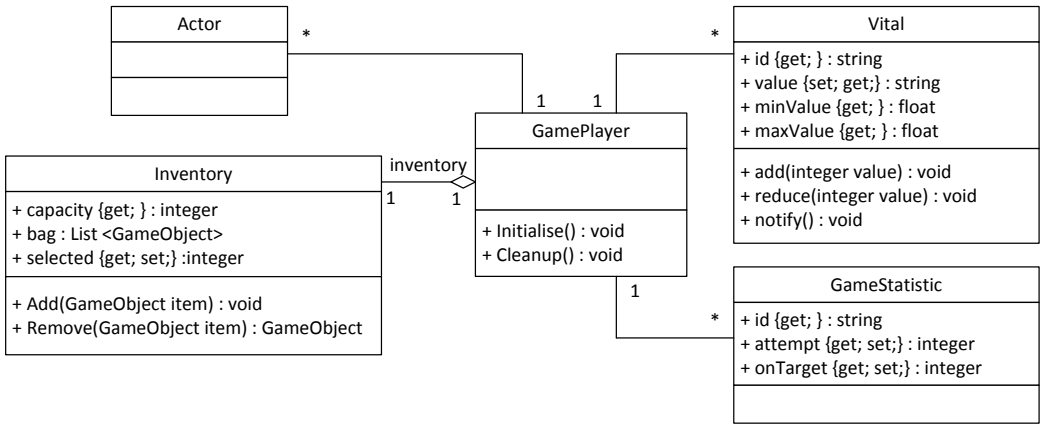

Figure B.3: UML Diagram for Game Player

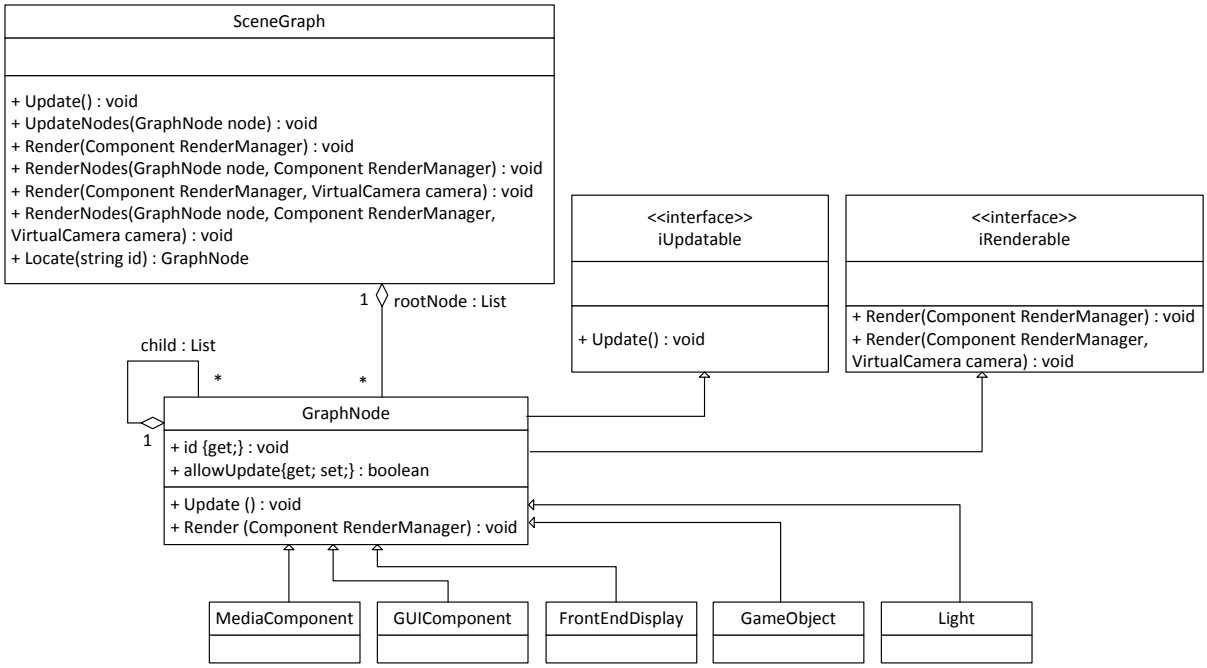

Figure B.4: UML diagram for scene graph – MediaComponent, GUIComponent, FrontEndDisplay, GameObject and Light inherit from the GraphNode.

```
Function Update()
{
	scenario.gameEnvironment.Update()
	2DGraph.Update()
}

Function Render(Component RenderManager)
{
	RenderManager.BeginRender()
	scenario.gameEnvironment.Render()
	2DGraph.Render()
	RenderManager.EndRender()
	RenderManager.RenderToDisplay()
}
```

Figure B.5: Example for updating and rendering scene graphs in the correct precedence in the simulation context.

```
Function Update()
{
	UpdateNodes(this.rootNode)
}

Function UpdateNodes(GraphNode node)
{
	node.Update

	For each GraphNode childNode in node.Child
	{
		UpdateNodes(childNode)
	}
}
```

Figure B.6: Example for recursively updating nodes in scene graph

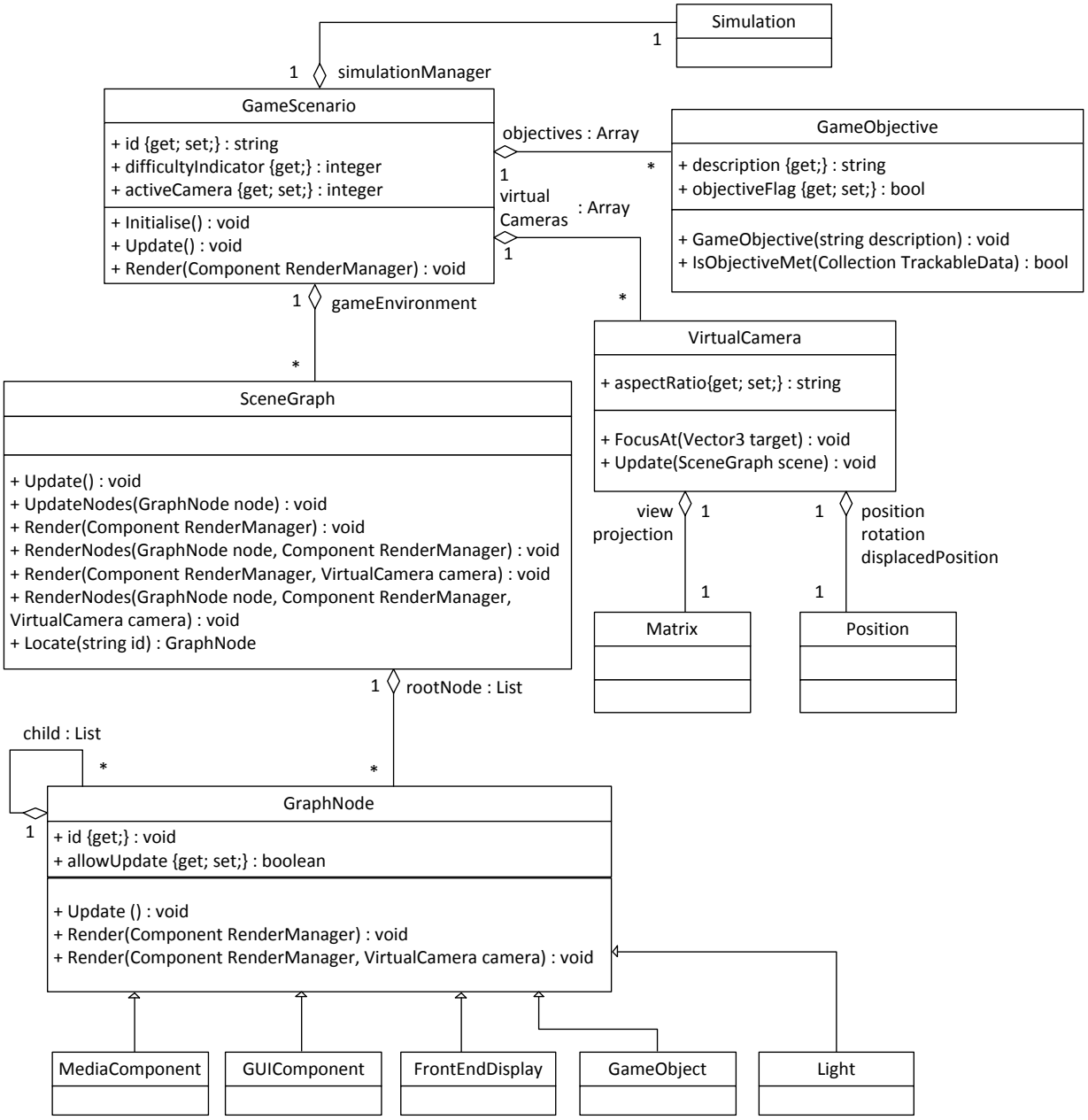

Figure B.7: UML Diagram for Game Scenario.

```
Function Update()
{
      //Check All Time Trigger
      Objective1Timer.CheckExpiry()

      //Trigger Game Event in the Scenario
      If(ActiveEvent == 1)
      {
            If(gameEnvironment.locate(actor1).activeState){ gameEnvironment.locate(actor1).activeState = 1}
            If(gameEnvironment.locate(actor2).activeState){ gameEnvironment.locate(actor1).activeState = 1}
            If(gameEnvironment.locate(actor3).activeState){ gameEnvironment.locate(actor1).activeState = 1}
      }
      ElseIf(ActiveEvent == 2)
      {
            If(gameEnvironment.locate(actor1).activeState){ gameEnvironment.locate(actor1).activeState = 2}
            If(gameEnvironment.locate(actor3).activeState){ gameEnvironment.locate(actor1).activeState = 3}

      }

      //Update Scene Graph
      gameEnvironment.Update()

      objectivesMet = true

      //Check to see if all objective are met
      For each objective in GameObjectives
      {
            objectivesMet = objectivesMet && objective.objectiveFlag
      }

      If(objectivesMet)
      {
            applicationEventManager.ReportEvent("ScenarioEnd")
      }
}
```

Figure B.8: Example of Update Method implementation for Game Scenario.

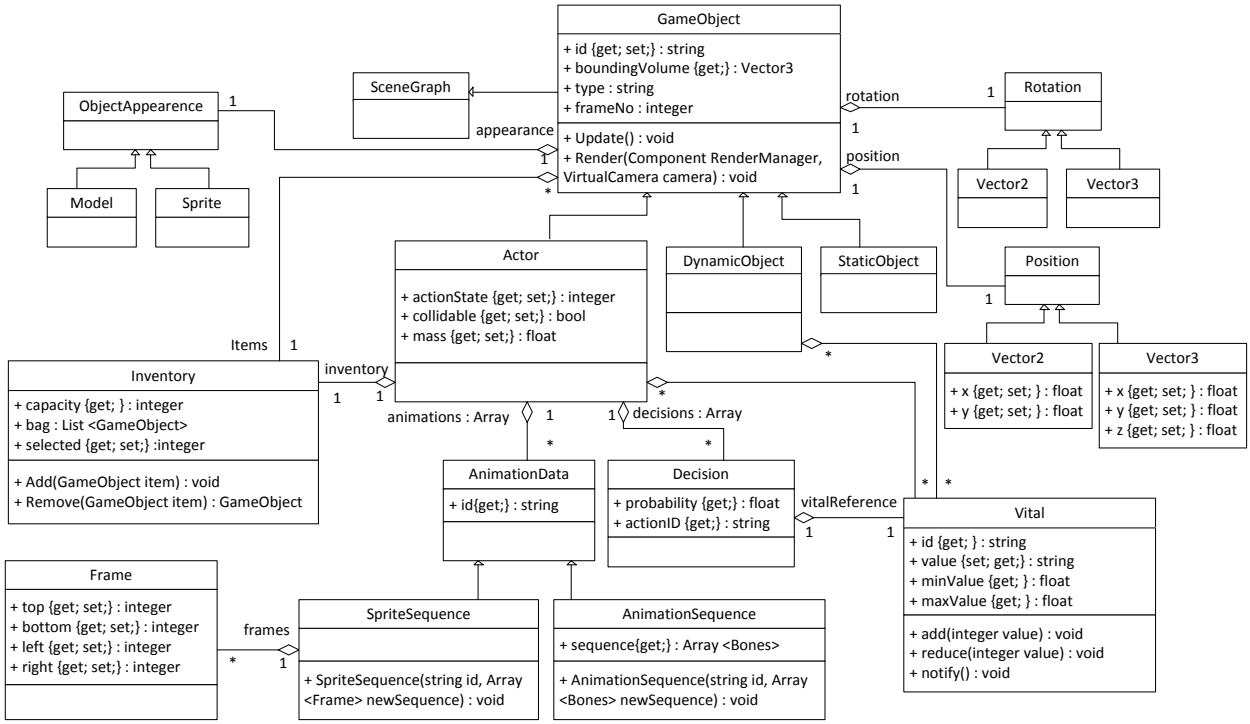

Figure B.9: UML Class Diagram for Game Object.

```
Function Update()
{
      //Pair action to the appropriate motion and animation
      If(this.actionState == 1)//Walk
      {
            collisionObject = PhysicManager.checkCollision(this, GameEnvironment)
            If(collisionObject.type == "Movable")
            {
                  PhysicManager.ApplyForce(this, 0.5f, this.rotation, PhysicManager.GetConstraint("Friction"))
                  PhysicManager.ApplyForce(collisionObject, collisionObject.mass/this.mass*0.5f,
                        this.rotation, PhysicManager.GetConstraint("Friction"))
                  AnimationManager.SetSequence(this, "Push")
                  SoundManager.Play(SoundManager.RetrieveSound("ActorPushSFX"), 1,
                     SoundManager.Event, this.position)
                  this.energy.reduce(0.5f)
            }
            Else
            {
                  PhysicManager.ApplyForce(this, 2.0f,this.rotation, PhysicManager.GetConstraint("Friction"))
                  AnimationManager.SetSequence(this, "Walk")
                  SoundManager.Play(SoundManager.RetrieveSound("ActorWalkSFX"), 1,
                     SoundManager.Event, this.position)
                  this.energy.reduce(0.1f)
            }
      }
      ElseIf(this.actionState == 2)//Pick
      {
            collisionObject = PhysicManager.checkCollision(this, GameEnvironment)
            If(collisionObject.type == "HealthPack")
            {
                  AnimationManager.SetSequence(this, "Pick")
                  this.health.add(25.0f)//health is a vital
            }
            ElseIf(collisionObject.type == "Key")
            {
                  AnimationManager.SetSequence(this, "Pick")
                  this.inventory.add(object);
            }
      }
      ElseIf(this.actionState == 3)//Use
      {
            If(this.inventory.selected.type == "Key")
            {
                  collisionObject = PhysicManager.checkCollision(this, GameEnvironment)
                  If(collisionObject.type == "HealthPack")
                  {
                        AnimationManager.SetSequence(this, "OpenDoor")
                        this.inventory.remove(this.inventory.selected)
                        SoundManager.Play(SoundManager.RetrieveSound("KeyUnlockSFX"), 1,
                           SoundManager.Event, this.position)
                        collisionObject.actionState = 1//Door Unlock
                  }
            }
      }
      Else{//Idle
            AnimationManager.SetState(this, "Idle")
      }

      //Check for collision
      collisionObject = PhysicManager.checkCollision(this, GameEnvironment)
      If(collisionObject.type == "Building" && collisionObject.type == "Furniture")
      {
            PhysicManager.ApplyRecoil(this, 0.1f)
            AnimationManager.SetSequence(this, "Stop")
      }

      //Update transform of animation
      AnimationManager.Update(this);

      //Reset actionState
      this.actionState = 0//idle
}
```

Figure B.10: Example of Update method implementation in a Game Object

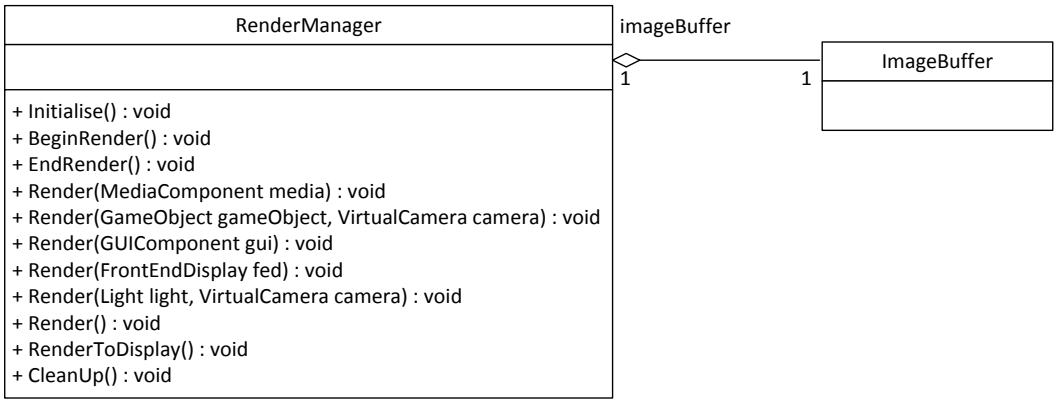

Figure B.11: Renderer represented in Class Diagram

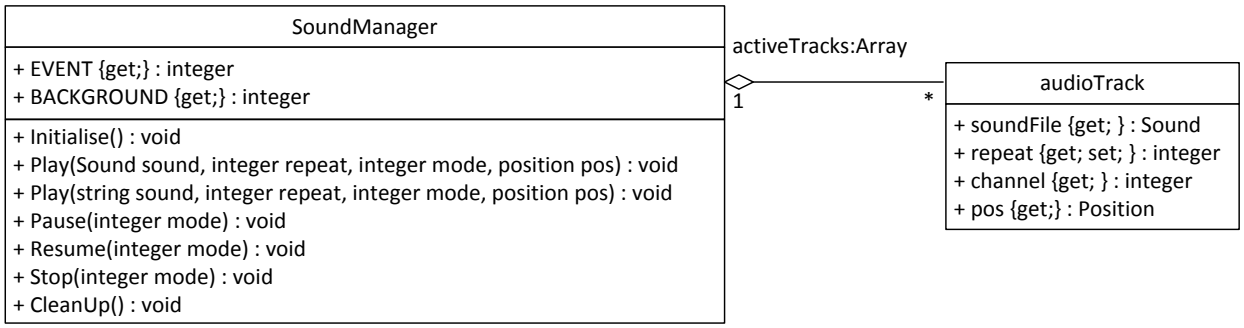

Figure B.12: Audio Component (SoundManager) in UML Diagram.

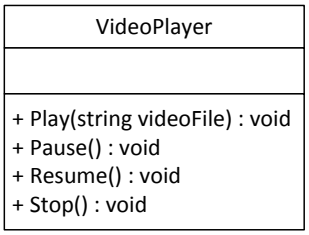

Figure B.13: Video Player Component (VideoPlayer) in UML Diagram.

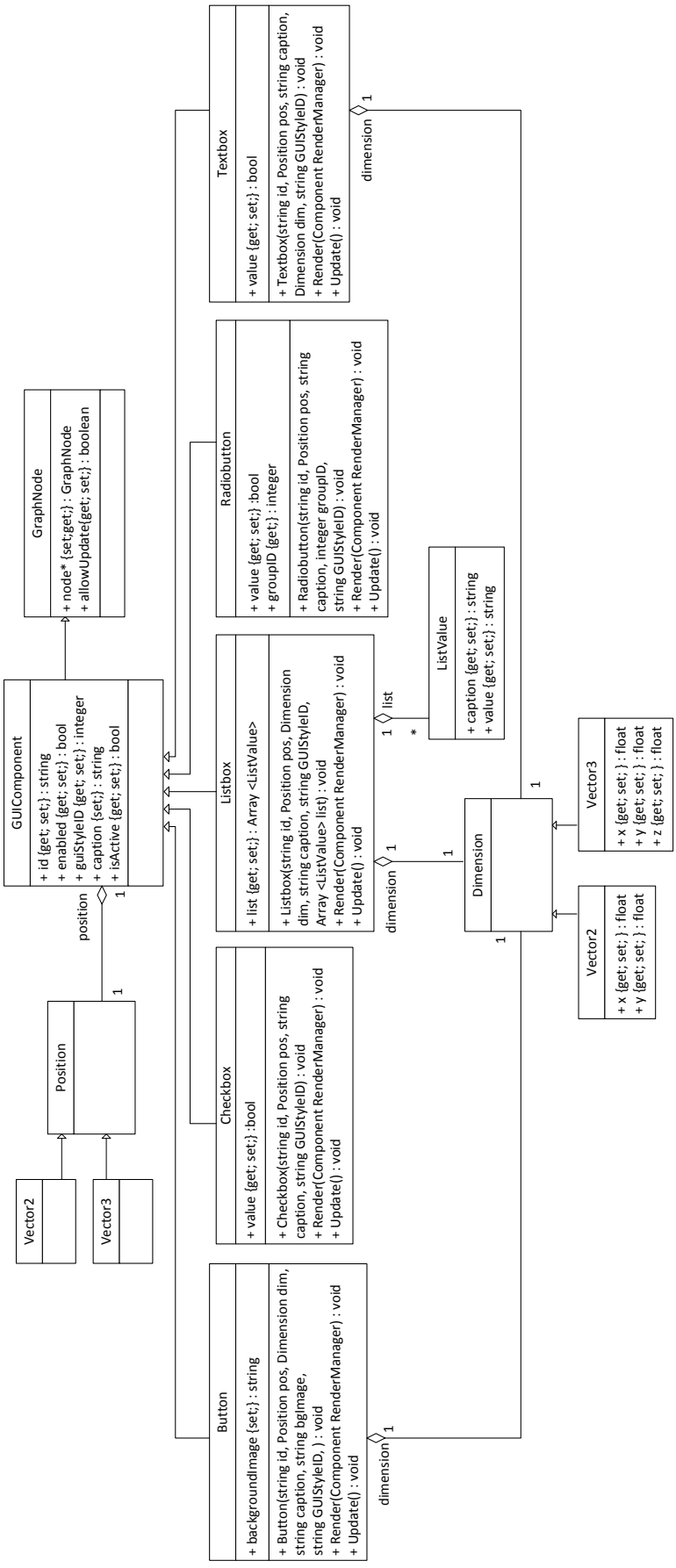

Figure B.14: GUI Component (GUIComponent) in UML Diagram.

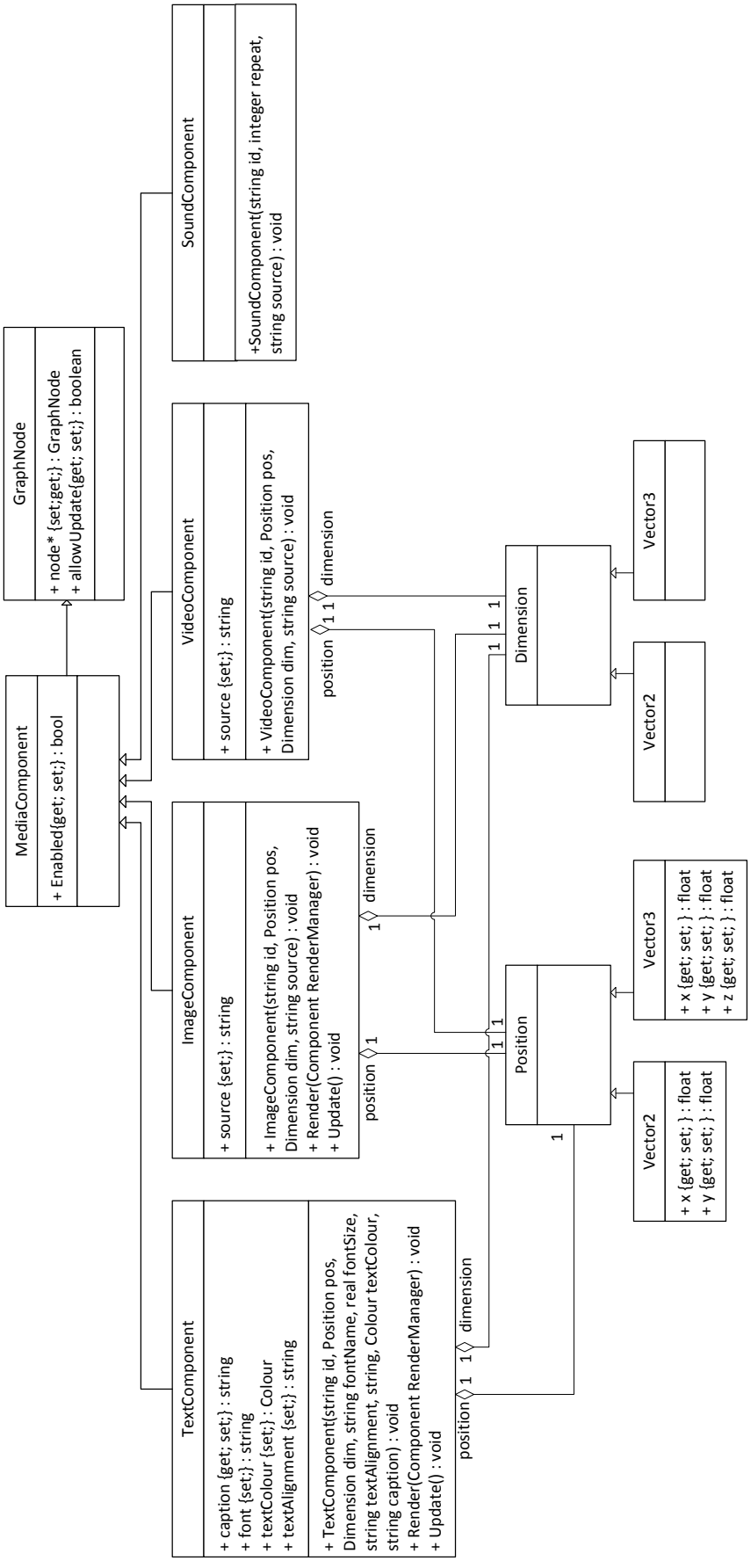

Figure B.15: Media Component (MediaComponent) in UML Diagram.

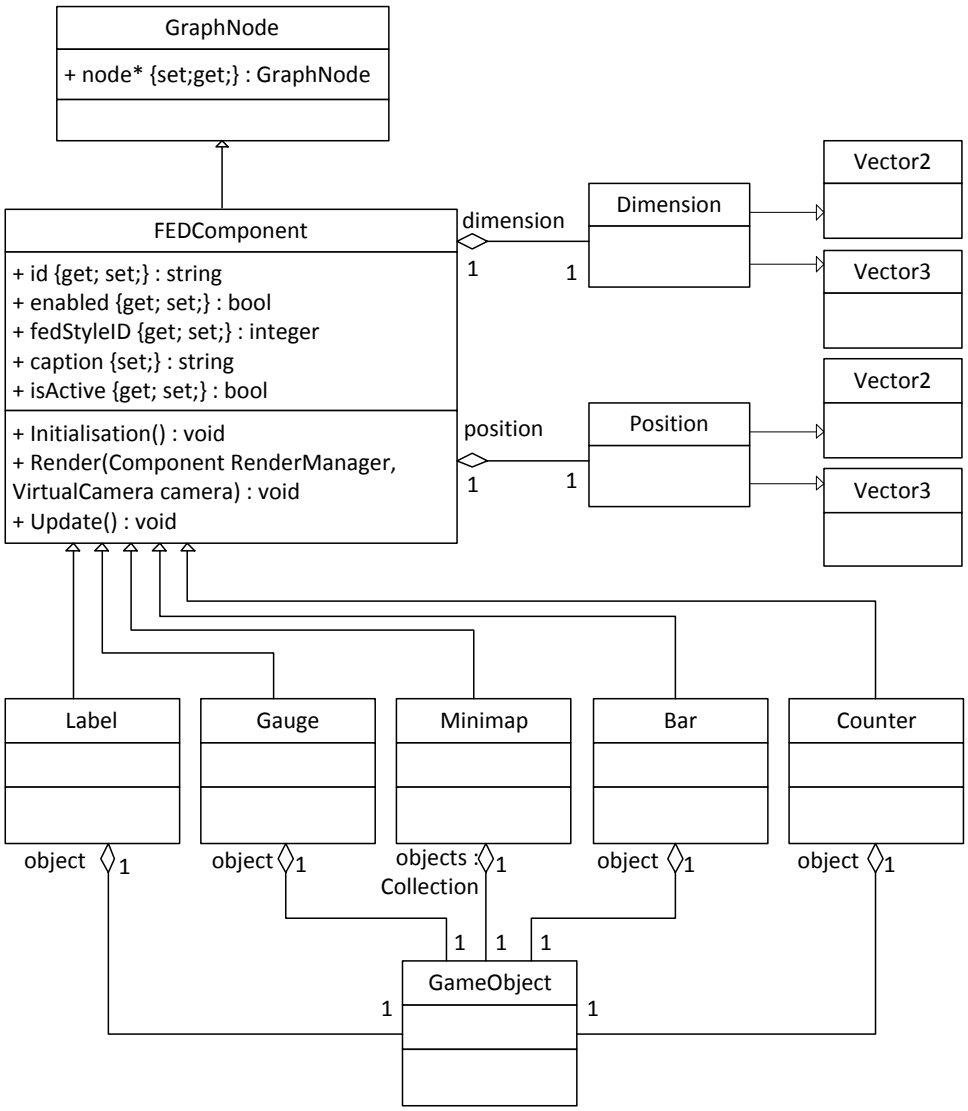

Figure B.16: Front End Display Component (FEDComponent) in UML Diagram.

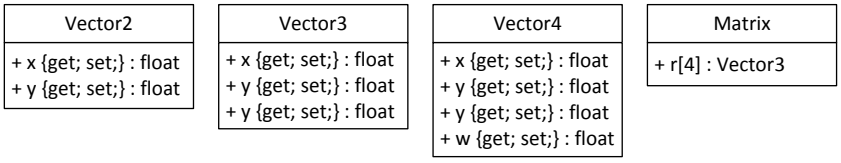

Figure B.17: Base classes for math library in UML Diagram.

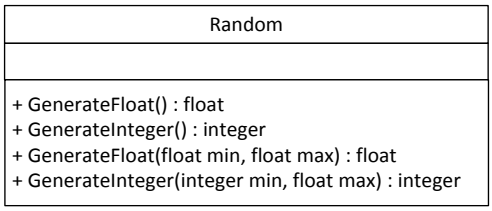

Figure B.18: Random number generator in UML Diagram.

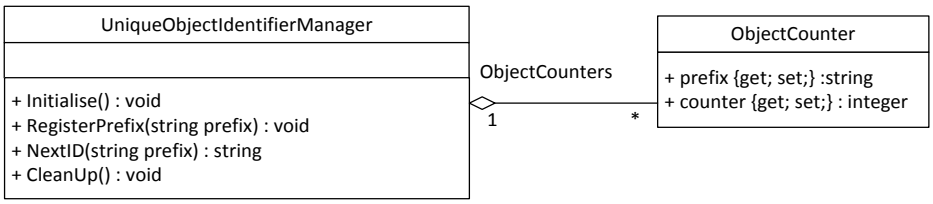

Figure B.19: Unique Object Identifier Management Component (UniqueObjectIdentifierManager) in UML Diagram.

APPENDIX C: Implementation of our Model Driven Game Development Framework to support the development of Serious Games

This appendix demonstrates the development of our novel model-driven serious game framework proposed in this thesis. In the following section, we present our prototype detailing the software architecture, the modelling environment, representation of model, and the translation and generation of model for the purpose of concept-proofing.

C.1 Overview of the Model-Driven Pipeline

The model-driven pipeline in our prototype is made-up of a modelling environment, model translators and an artefacts generator. We choose to implement our own set of tools instead of using the MDE tools described in Section 3.7 mainly due to the reason choice of platform we wish to deploy the modelling environment and to generate the artefacts. In addition, it also provides us the flexibility to develop a less technical modelling environment for the non-technical domain expert. An overview of this is shown in Figure C.1.

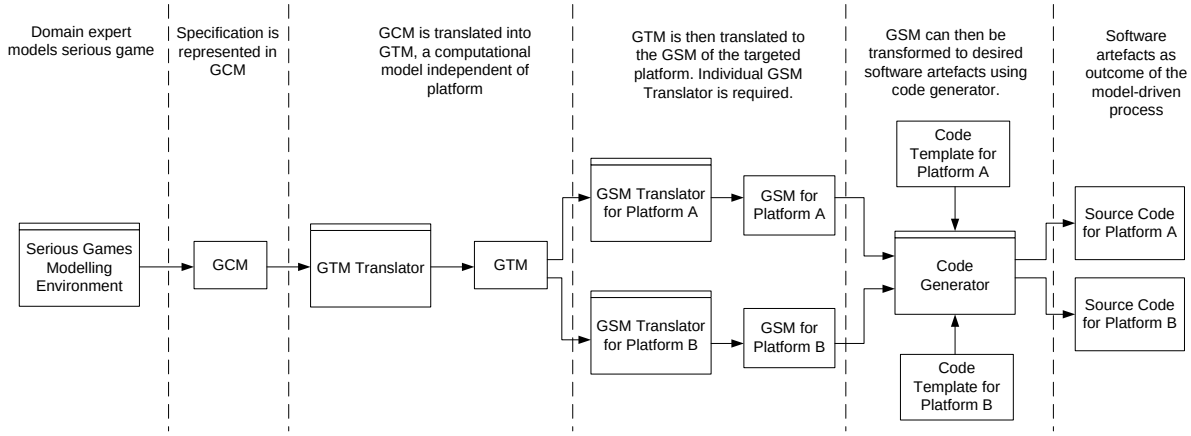

Figure C.1: Model-driven pipeline for the prototype

The modelling environment is a GUI-based environment which allows non-technical domain expert to model serious game using both graphical notations and step-by-step wizard to produce a serious game model that is compliant to our Game Content Model. We have chosen to develop a web-based modelling environment due to the wide-access the web can offer to the game-based learning community. This approach can also lower the barrier of entry for adoption of game-based learning as practitioners no longer require a high performance multimedia computer to produce serious game. The Adobe Flash platform is chosen as the development platform of the modelling environment

because it offers the facilities to support the development this modelling environment with rich interactive features and it can allow most users with connected desktop to use this tool without requiring any local installation. Our rich experience of using Adobe Flash platform is also one of the main reasons behind this choice as it allows us to rapidly prototype and demonstrates the user interface aspect of our model-driven serious game development framework. Non-technical domain experts will be using this modelling tool to author the serious game content collating the art assets and defining the necessary game mechanics that made up the serious game.

The serious game design (Game Content Model) produced using the modelling environment is then represented using a middleware which is later translated into other models using the model transformation tool. In this prototype, we have chosen to use eXtensible Markup Language (XML) as the middleware for model representation. This eases the transformation process and allows additional information to be weaved into the later models.

The Game Content Model will then be translated to a Game Technology Model using a MDE tool. In this prototype, we have implemented a simple model translation tool in PHP to allow transformation to be processed at the server-end instead of using the MDE tools as described in Section 3.7. PHP is chosen amongst the available server-side technology because it is an open platform and it provides the facilities we required to process the Game Content Model generated from the SeGMEnt tool. The Game Content Model is read by the model translation and the model is transformed into Game Technology Model.

The next stage in our model driven pipeline, the Game Technology Model is processed by the next model translator which weaved in the necessary platform specific information to create a platform specific model referred to in our framework as Game Software Model. This Game Software Model translator is also implemented in PHP. Since our case studies will be targeting the serious game design to the Adobe Flash platform, the Game Software Model will be designed to include necessary platform specific information to the Game Technology Model. This means a separate Game Software Model is needed if a different technology platform is targeted.

Finally, we will be generating software artefacts using the code generator and appropriate code template.

C.2 Serious Games Modelling Environment (SeGMEnt)

Our modelling environment, referred to as Serious Games Modelling Environment (SeGMEnt), is designed to allow non-technical domain expert to document serious game design formally. In general

the task of modelling serious games components can be categorised into *data modelling* and *visual modelling*. *Data modelling* mostly involves definition of objects, flows and processes whereas *visual modelling* involves positioning the in-game components in the virtual world, constructing the virtual environment and arranging the GUI components on-screen. In some aspects of the serious games modelling, visual modelling is crucial to enable domain expert to visualise the information to be used in data processing. Although a portion of serious games design will involve organisation of visual elements, this framework will focus mainly on visual modelling that provides positional data of in-game components that is useful in serious games design. Our SeGMEnt tool encapsulates all the concepts of Game Content Model into the different design viewpoints namely *structure*, *object*, *simulation*, *presentation*, *environment* and *player* using the appropriate UI model.

C.2.1 Architecture for SeGMEnt in Adobe Flash

Our SeGMEnt tool is implemented in Adobe Flash using ActionScript 2.0 platform. This is mainly due to our extensive knowledge in the ActionScript 2.0 API and deep understanding of the working of ActionScript 2.0. The modelling environment itself consist of three parts namely (1) the *shell* which stores all the data structures holding the data representing the serious game model-being-model, (2) the *collapsible panel* which consist of controls that allow users to switch between viewpoints and additional controls for each viewpoints, and finally (3) the individual *viewpoints* which allow users to model specific aspect of a serious game. Each of this part has been manually programmed to reside on a different level of the Adobe Flash Player movie stack. The shell takes the base-level of the stack namely *_level0* and controls the loading and unloading of viewpoints and panel. All viewpoints are dynamically loaded onto *_level1* and collapsible panel is placed at *_level2* to allow the controls to float above the active viewpoint as shown in Figure C.2.

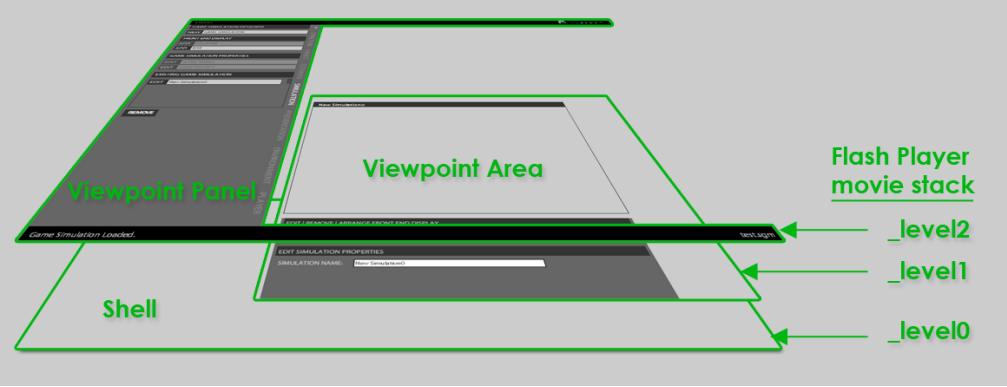

Figure C.2: Architecture of SeGMEnt in Adobe Flash

C.2.2 User Interface (UI) Components for SeGMEnt

In the design of our SeGMEnt, we have identified five unique UI components that can assist non-technical domain experts when modelling the aspects of serious games design in the SeGMEnt modelling environment. These UI components are *flow visualisation, dynamic option interface, What-you-see-is-what-you-get (WYSIWYG) visualisation, statement construction interface* and *guided data entry interface.*

C.2.2.1 Flow Visualisation

The flow visualisation is the UI component responsible for providing user with the visualisation that represents the flow of the serious game. In this visualisation component, we have chosen specifically to implement a state diagram notation that has been extended to include additional information. Each of the diagram would consist of a begin state, an end state and the in-between states. Each state (referred to as context in our Game Content Model) would consist of one or more transitional conditions to the next state depending on specification of each state. As a UI component, each state can be repositioned while maintaining the link(s) with other states that it is related to. To ensure that users are not overloaded with information in a single screen, the additional information associated to each state is encapsulated within the state and can be viewed on a separate view. Whenever is possible a Dynamic Option Interface (refer to Section C.2.2.2) is used to allow domain expert to select the options available instead of a textbox. These options are data which have been retrieved from other views to prevent error in data entry. The breakdown of flow visualisation elements are is illustrated in Figure C.3.

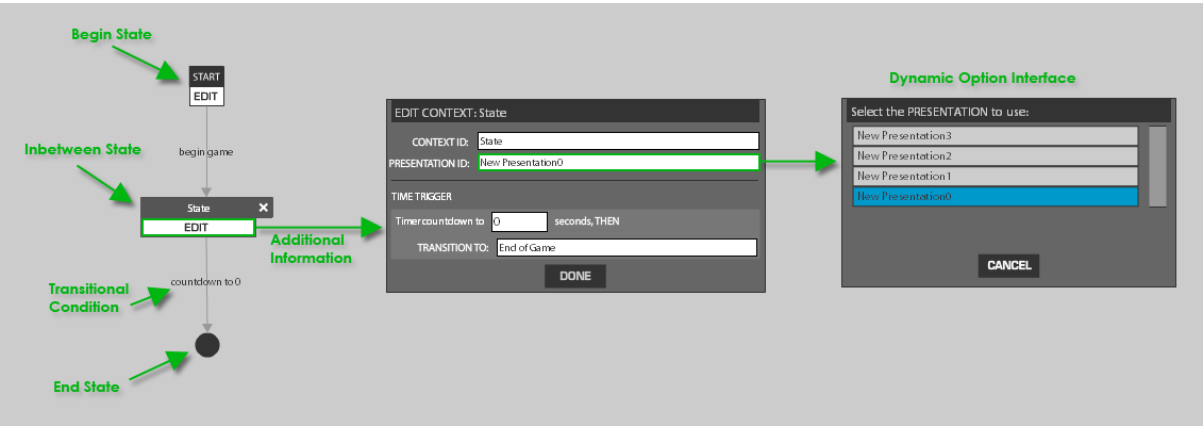

Figure C.3: Elements of Flow Visualisation UI

One of the biggest challenges when implementing this UI in Adobe Flash ActionScript 2.0 is the drawing of line that represents the transitional condition between source-state and target-state. This visual representation of transitional condition needs to be maintained regardless of the position of the source-state or target-state. Our solution to this problem is to maintain two collections of data structures; (1) a collection for storing the data for each individual state and (2) a separate collection for storing all the transitional condition as shown in Figure C.3. This allows us to easily distinguish data from visual aids (transitional condition arrows). The transitional condition arrows are updated based on user's interaction through a fixed-time interval (onEnterFrame event handler) redrawing operation (See Figure C.4 for code snippets).

```
canvas_mc.gStructure_mc.attachMovie("startSymbol_mc", _context, -1, {_x:_ox, _y:_oy});
canvas_mc.gStructure_mc[_context]._contextID = "Start Game";
canvas_mc.gStructure_mc[_context]._type = "start";
canvas_mc.gStructure_mc[_context]._nextContext = _nextContext;
canvas_mc.gStructure_mc[_context]._nextContextID = _nextContextID;
canvas_mc.gStructure_mc[_context].header_btn.onPress = function(){
    if(!editFlag){
        this._parent.startDrag();
        this._parent._alpha = 50;
        canvas_mc.gSRelationship_mc.onEnterFrame = function(){
            drawRelationships_fn();
        };
    }
};
```

Figure C.4: ActionScript 2.0 snippet that represents data structure for the Start Symbol and the onPress event handler which triggers the fixed-time interval redrawing operation when symbol is in not in editing additional information mode.

```
function drawRelationships_fn(){
    for(var i = 1; i < canvas_mc.gSRelationship_mc._index; i++){
        if(canvas_mc.gSRelationship_mc["relationship" + i + "_mc"] != undefined){
            canvas_mc.gSRelationship_mc["relationship" + i + "_mc"].draw_fn();
        }
    }
}
```

Figure C.5: ActionScript 2.0 snippet that shows how transitional information arrows (`gSRelationship_mc`) collections are traversed and redraw (`draw_fn()`) operation is invoked.

In SeGMEnt, flow visualisation component used in *Structure Designer* for modelling structure and flow of serious game and in *Scenario Designer* for modelling sequences of events in a game scenario.

C.2.2.2 Dynamic Option Interface

We recognised that users are required to enter data that refers to an aspect of the serious game design in different viewpoints throughout the modelling process. The dynamic option interface, as previously

mentioned, is a list of option generated from data fetched from within SeGMEnt during runtime to simplify the data entry process and to prevent error in data entry. We have implemented two versions of the dynamic option interface; (1) a fixed dynamic option interface which can be part of a data entry interface and (2) a floating dynamic option interface which acts as a substitute to the traditional list GUI (see Figure C.6).

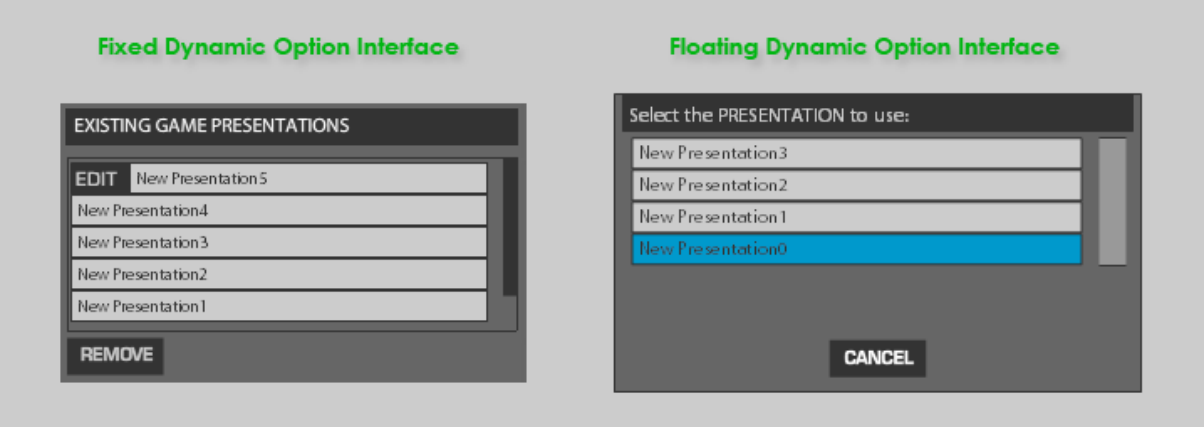

Figure C.6: Floating Dynamic Option Interface and Fixed Dynamic Option Interface.

The fixed dynamic option interface provides customisable options to add and remove option from the list whereas the floating dynamic option interface only shows options related to the specific entry. The options for each dynamic option interface are maintained as an array of data structure. Adding and removing option from the fixed dynamic option interface are achieved by updating the array either by pushing in new option or removing the data structure from a specific index of the array. While the fixed dynamic option interface allows users to add and remove options via GUI controls such as button, the options for floating dynamic option interface are constructed from existing collection of data structure within the same viewpoint or from other viewpoint. Both the fixed and the floating dynamic option interfaces are used widely in different viewpoints of SeGMEnt.

C.2.2.3 WYSIWYG (What-you-see-is-what-you-get) Visualisation

The WYSIWYG visualisation is a basic UI component that aids users to visually position media, GUI, FED and game objects strategically on a 2D space. In WYSIWYG visualisation, the media, GUI, FED and game objects are represented as symbols which can be drag and drop to the desired position on the screen and the visual properties of these can be edited via data entry interface (see Figure C.7). Each 2D space maintains an array that stores data structure representing visual properties of the relevant visual symbols which is saved and loaded into the relevant viewpoint at any point of the modelling process. The WYSIWYG visualisation is used in the presentation viewpoint to allow user to

model presentation context, the scenario viewpoint for modelling game interfaces and the environment viewpoint for modelling of game environment which will be used in a game scenario.

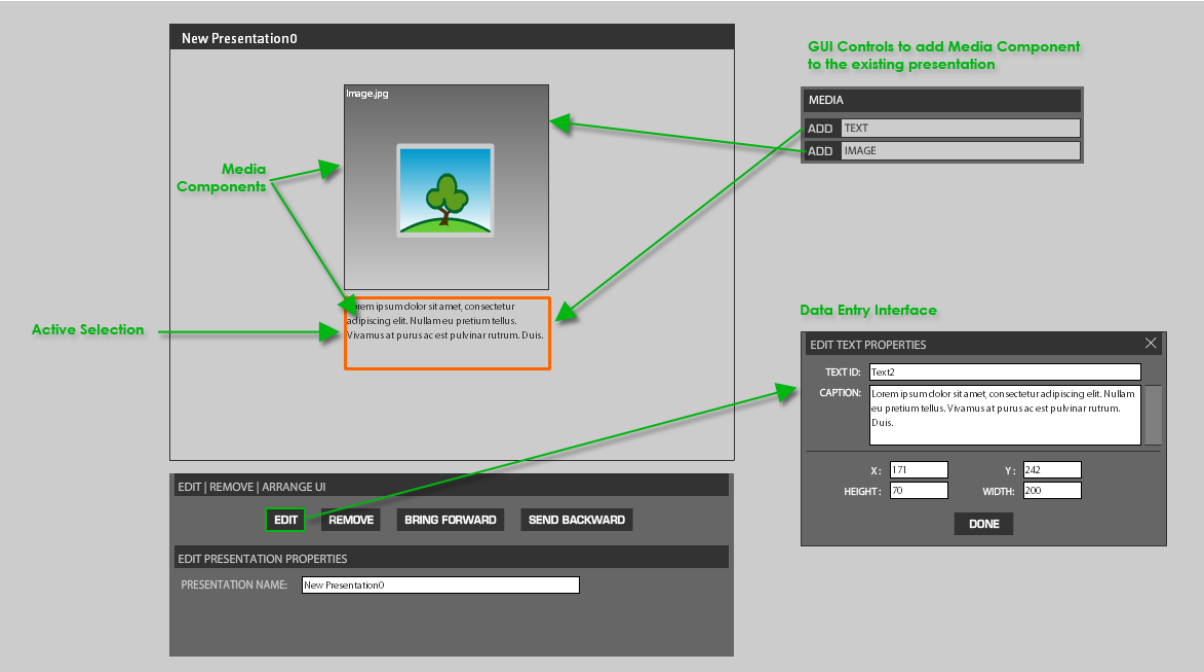

Figure C.7: WYSIWYG Visualisation in Presentation Designer viewpoint.

C.2.2.4 Statement Construction Interface

The statement construction interface provides assistance for users to construct a statement by selecting the verb, noun or conjunction from the options presented in floating dynamic option interface. In our SeGMEnt tool, the statement construction interface is used primarily to guide users in constructing acting statement which is an English-like sentence that define game act (see Section 5.3.6 on use of game act in game event) that is executed in the arranged order. This allows user to provide the necessary information for executing specific actions to represent the scenario similar to the role of a director giving instructions to the actor. Constructions of act statements are constrained by grammar of act statement as shown in Table C.1.

Object (in-game object), *action* (behaviour of in-game object), *speech* (text or sound of a speech) and *location* (a marked position in the virtual space) are nouns, and *act*, *speak* and *moveTo* are the available verbs use to construct simple act statement which begins with subject then follow with predicate based on a declarative sentence structure in English. Compound act statement supports the use of AND conjunction to create a more complex act. Act could be interrupted in the game-play when game-player interacts with in-game object during execution of an act sequence as part of reaction of in-game component and resume after that.

Table C.1: Grammar for Act Statement.

Descriptor	Syntax
Noun	<object> | <action> | <speech> | <location>
Verb	act | speak | moveTo
Conjunction	AND
Simple Act Statement	<simple act statement> = <subject> <predicate> <subject> = <object> <predicate> = act <action> | act <action> <object> | speak <speech> | moveTo <location>
Compound Act Statement	<compound act statement> = <subject> <predicate> AND <predicate> [AND <predicate>] <subject> = <object> <predicate> = act <action> | act <action> <object> | speak <speech> | moveTo <location> Constraints: <object> act <action> AND speak <speech> | <object> act <action> <object> AND speak <speech> <object> act <action> AND speak <speech> AND moveTo <location> | <object> act <action> AND moveTo <location>

In our implementation, a word is chained to another set of options depending on the type of word selected whether it is a noun, verb or conjunction. Chaining of words is depending on the grammar described in Table C.1. This is programmed in the *nextWord_fn()* function which is invoked after an option has been selected (see Figure C.8). This will add the next word to the statement and have it positioned visually in the correct position as shown in Figure C.9.

```
function addActor_fn(mc:MovieClip, _caption:String):String{
    var mcName = "Actor_mc";
    mc.attachMovie("word_mc", mcName , 1);
    mc[mcName]._x = 10;
    mc[mcName].caption_txt.text = _caption;
    mc[mcName].caption_txt.autoSize = true;
    mc[mcName].bg_mc._width = mc[mcName].caption_txt._width;
    mc[mcName].bg_mc.onRelease = function(){
        restartSentenceFrom_fn(this._parent);
        disableCoordinations_fn(editActingScript_mc, this._parent._parent._name);
        this._parent._parent.swapDepths(this._parent._parent._parent.getNextHighestDepth());
        this._parent.attachMovie("list_mc", "list_mc", 1, {_x:-40, _y:-50});
        this._parent.list_mc._buttonSelected = this._parent.caption_txt.text;
        this._parent.list_mc._maxItem = 6;
        this._parent.list_mc._index = 0;
        this._parent.list_mc._caller = "";

        var options:Array = actorList;
        buildList_fn(this._parent,this._parent.list_mc, options);
        initList_fn(this._parent,this._parent.list_mc, options, "Select a SUBJECT");
    };

    mc[mcName].nextWord_fn = function(){
        this._parent.swapDepths(this._parent._oDepth);
        this.caption_txt.autoSize = true;
        this.bg_mc._width = this.caption_txt._width;
        addCommand_fn(mc, mcName, "Select a COMMAND");
    };
    return mcName;
}
```

Chaining of word using `nextWord_fn()` function

Figure C.8: Code snippet showing how words are chained one after another according to the grammar of acting script in ActionScript 2.0

Figure C.9: Statement Construction Interface.

The statement construction interface is used in the scenario designer viewpoint to allow users to specify acting statement for an event as shown in Figure C.10.

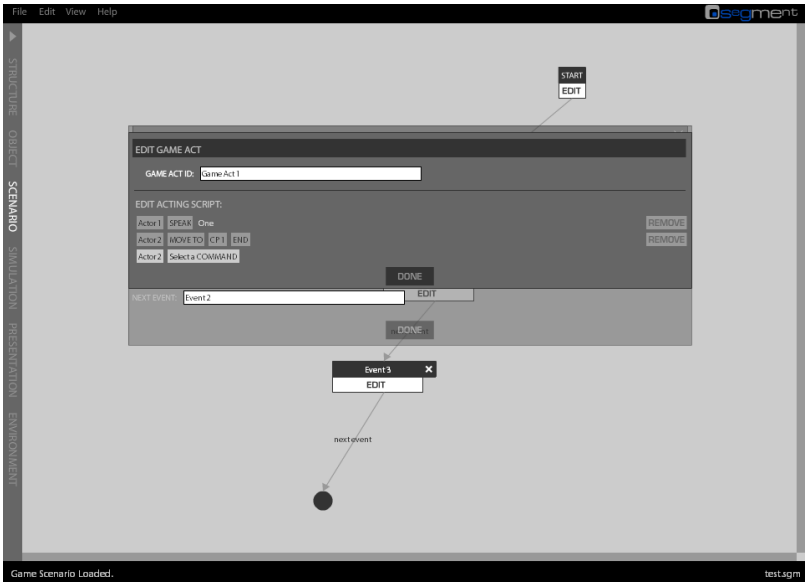

Figure C.10: Statement Construction Interface in Scenario Designer Viewpoint.

C.2.2.5 Guided Data Entry Interface

The guided data entry interface provides a step-by-step guide for users to document aspects of serious games in our SeGMEnt tool systematically and to avoid overloading request of information from users. Each step in the guided data entry interface can be regarded as a tab control that groups a subset of data entry task into a single unit. GUI such as textbox, radio button, dynamic option interface and button are used to form the data entry UI for each step. Each of the steps has had the UI manually coded and data which are linked with other viewpoints are fetched automatically during runtime to be presented as options in the dynamic option interface (see Figure C.11). In SeGMEnt, the guided data entry interface is used in the object designer viewpoint to guide users in defining game objects and the player designer viewpoint for defining aspects of game player.

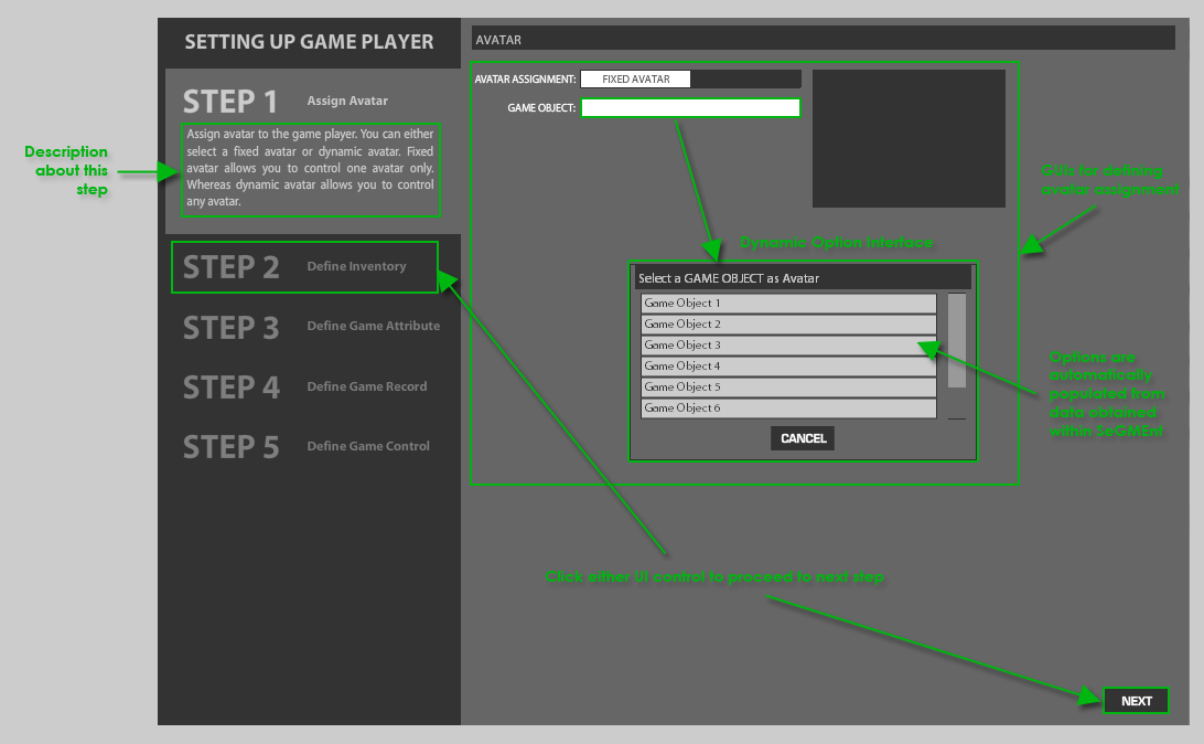

Figure C.11: Guided Data Entry Interface in Player Designer Viewpoint.

C.2.3 Design Viewpoints in SeGMEnt

The viewpoints in our SeGMEnt tool are designed to separate aspects of serious games design into smaller and manageable clusters of data set. Each of the viewpoints uses a combination of UI components described in Section C.2.2 to aid users in visualising aspects of serious game design. In the following subsections, we will describe the viewpoints and explain how our Game Content Model is encapsulated within the UIs and viewpoints.

C.2.3.1 Game Structure Designer

The game structure designer is the viewpoint where users model the flow and the structure of a serious game which is described in Section 5.3.1. In the game structure designer users specify the type of context, complete the details of transitional condition and the next context in the flow. Some of these data entry task are made easier through the dynamic option interface where users are only required to select from a list of available options (see Figure C.12). As described before, these options are automatically generated from data gathered from different viewpoint with the aim to prevent data inconsistency and simplify the interaction. The systemic view of the game structure is presented using the flow visualization UI component as shown in Figure C.13.

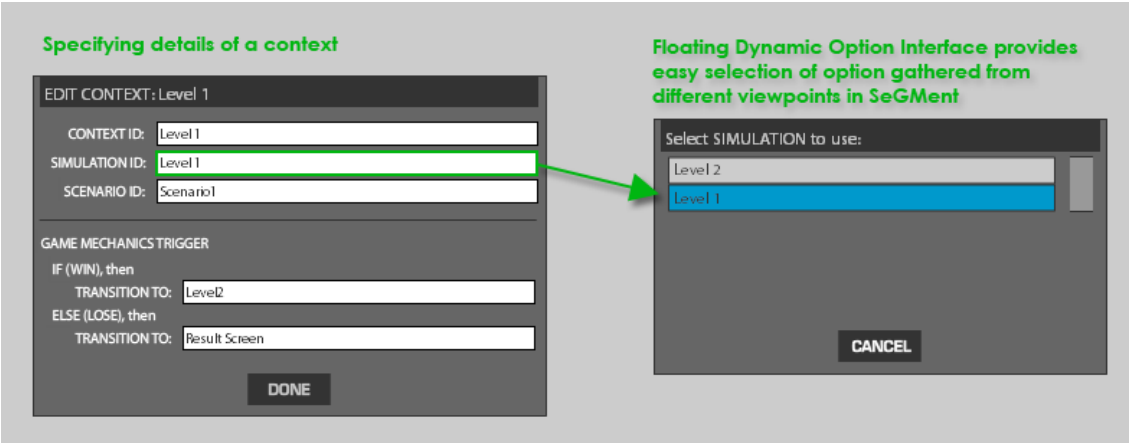

Figure C.12: Editing context in Game Structure Designer with the aid of Dynamic Option Interface.

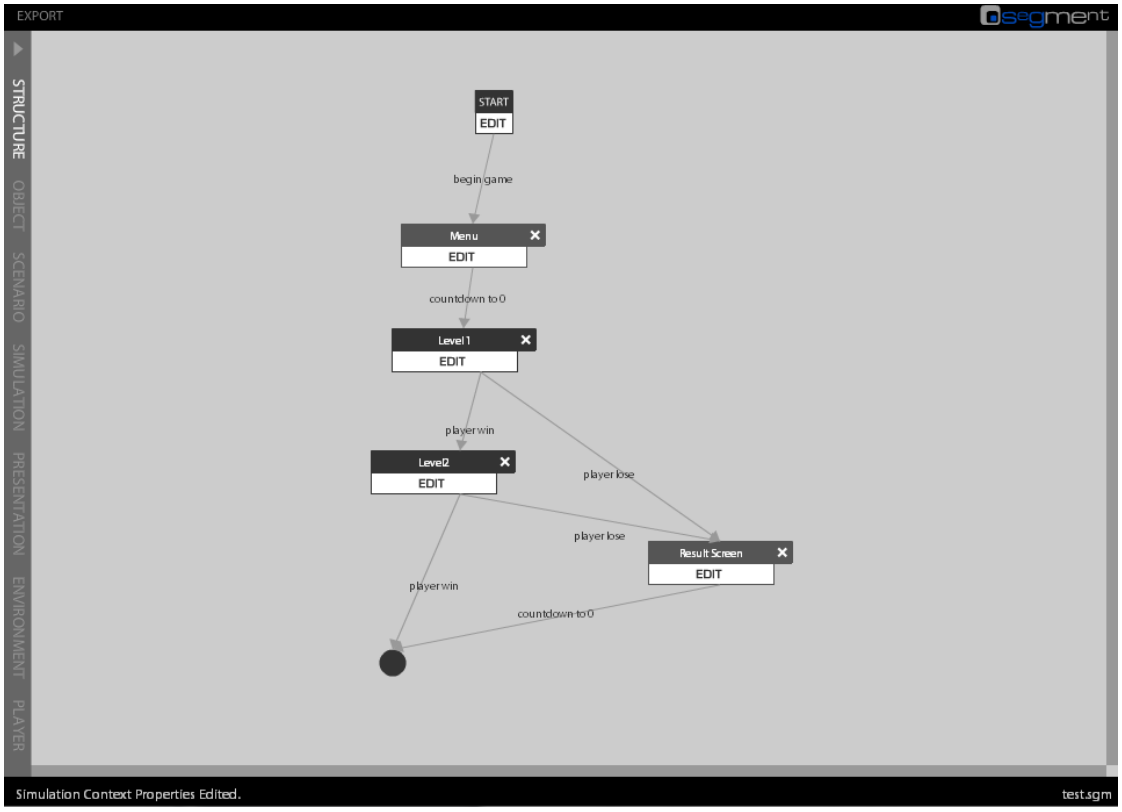

Figure C.13: Modelling Game Structure in Game Structure Designer.

C.2.3.2 Game Scenario Designer

The game scenario designer is the viewpoint where users specify the flow of events in a scenario. This viewpoint allows user to describe the game scenario (refer to Section 5.3.5), game event (refer to Section 5.3.6) and game objective (refer to Section 5.3.7) in our Game Content Model. Within each event, domain expert can specify the game acts that compose the event similar. Each game act is composed using ActingScript. In our modelling environment we have the grammar of the ActingScript built into the UI where domain expert can select the verb and noun to construct a valid

game act which we have described in Section C6.2.2.4 (see Figure C.14). Each of the game act created is added to the list of existing game act library which users can reuse in other game events (see Figure C.15). In addition to game act, domain experts are required to define the game rules in which a scenario operates under. Domain experts can define this through a UI with the use of dynamic option interface as shown in Figure C.16.

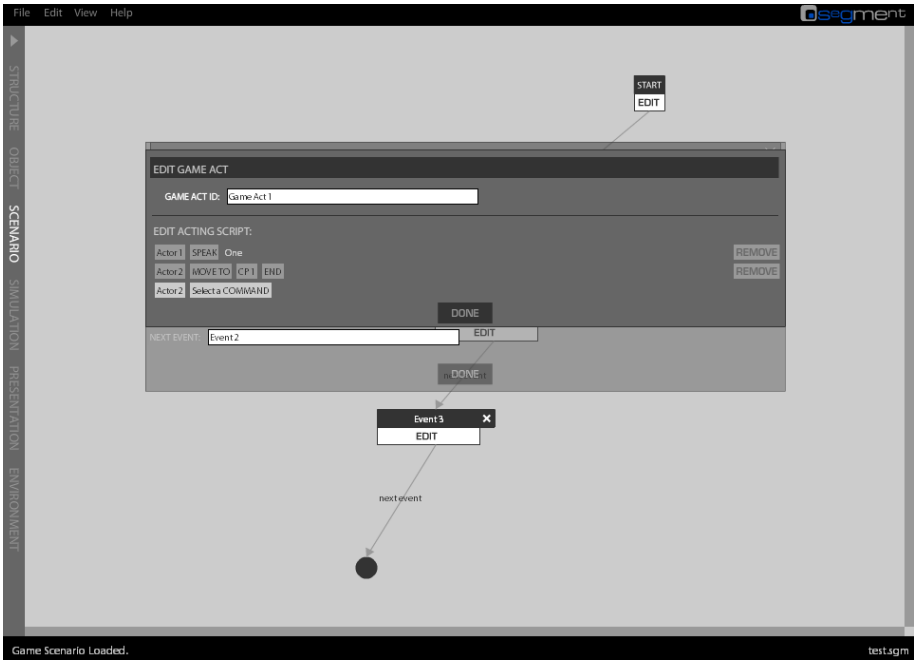

Figure C.14: Editing game act using ActingScript UI in Game Scenario Designer.

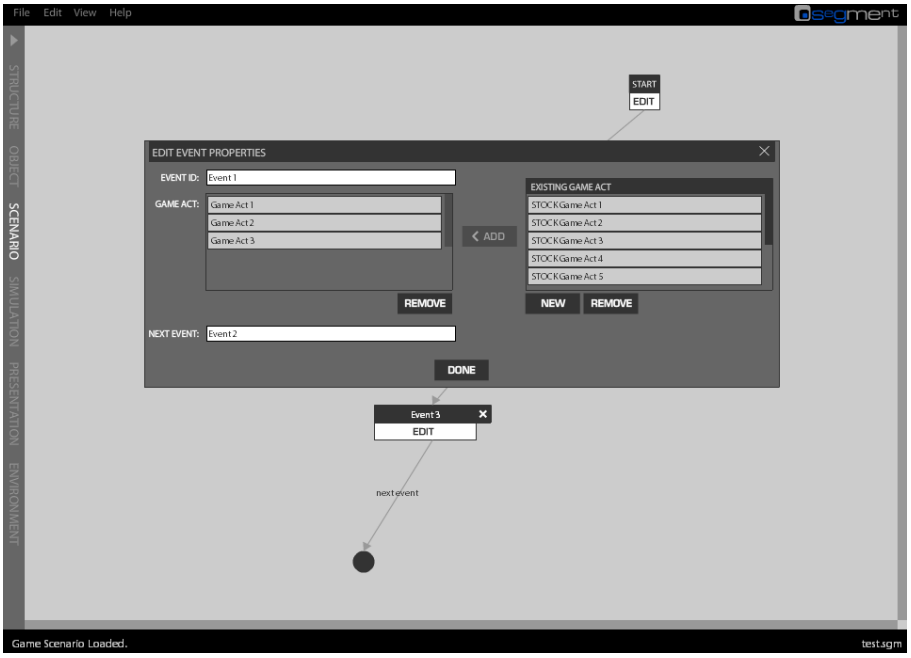

Figure C.15: Each game act created is added to the existing game act library where user can reuse in other game events.

Figure C.16: Defining game rules in Scenario Designer.

In this game scenario designer viewpoint, users can also define game objectives through the UI shown in Figure C.17 with assistive support of dynamic option interface.

Figure C.17: Defining game objective in Game Scenario Designer.

C.2.3.3 Game Object Designer

The game object designer is the viewpoint for users to define a game object's identity, specify the associated attributes, assign an appearance, define actions and define associated intelligence which is describe in game object concept in our Game Content Model (see Section 5.3.4). In this design viewpoint, we have chosen to use the guided data entry UI component to aid users the process entering data. Likewise what have been implemented throughout the SeGMEnt modelling experience, we retrieve data from other viewpoints to ease the entry of data by the domain expert using the dynamic option user interface.

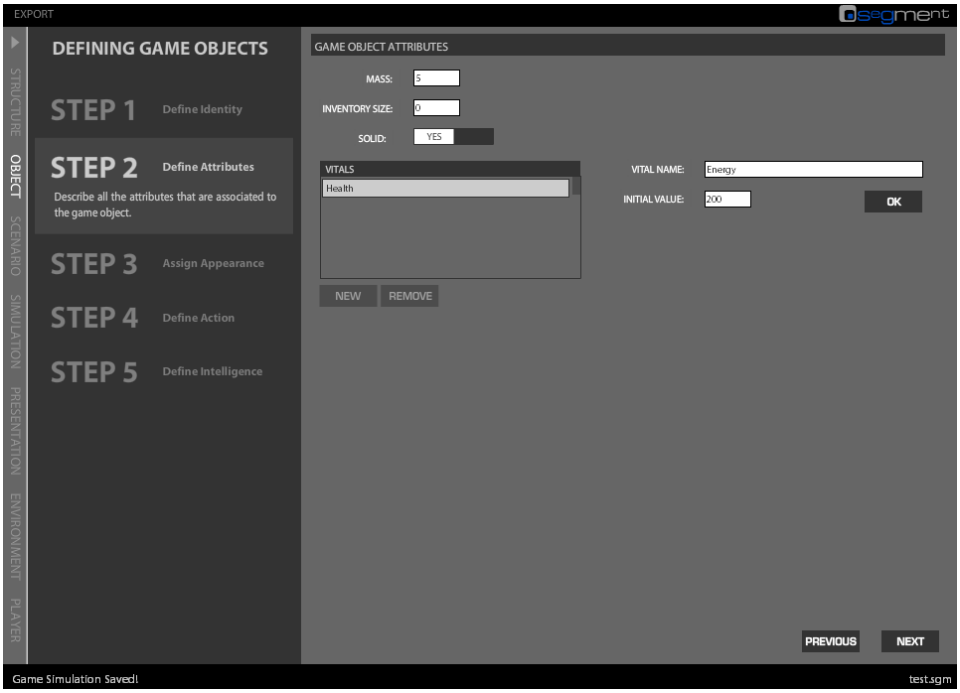

Figure C.18: Defining game object in Game Object Designer.

C.2.3.4 Game Simulation Designer

The game simulation designer is designed based on the game simulation concept described in Section 5.3.3. It is the viewpoint for users to add FEDs to the screen, drag it to the right position and edit the associated properties through the WYSIWYG Visualisation (see Figure C.19). In addition to the game user interface design, user can define the game tempo and physics through a dialogue box simply by providing the necessary information (see Figure C.20).

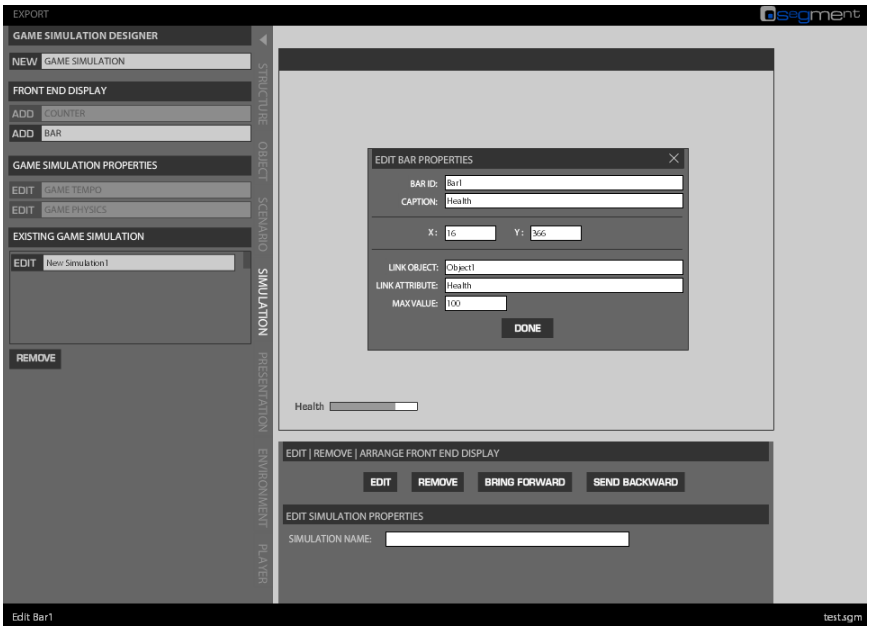

Figure C.19: Designing game interface in Game Simulation Designer.

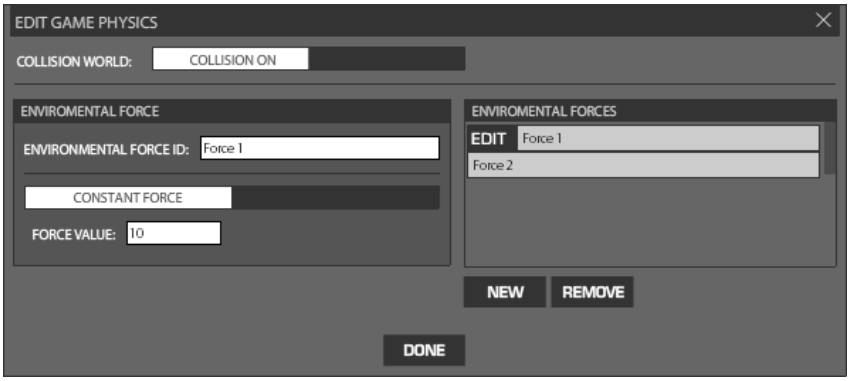

Figure C.20: Defining game physics in Game Simulation Designer.

C.2.3.5 Game Presentation Designer

The game presentation designer offers the viewpoint for users to model off-game user interfaces such as a menu screen, a screen that presents game player with the game objectives and a screen that present the result to game player. This viewpoint is designed based on the game presentation concept in our Game Content Model (see Section 5.3.2). It uses the WYSIWYG visualisation which allows users to drag each media or GUI components into position and edit their properties through the dialogue box (see Figure C.21). The presentation context created will then be referenced in the structure designer viewpoint for modelling of game flow.

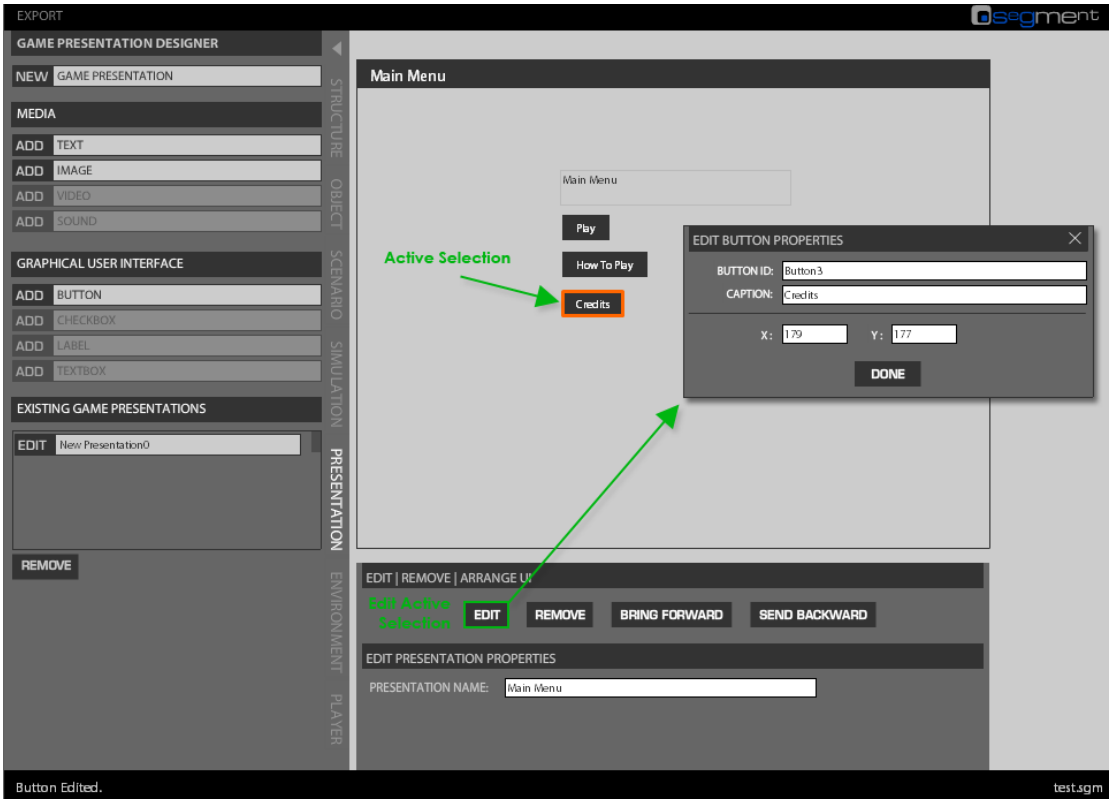

Figure C.21: Modelling game presentation in Game Presentation Designer.

C.2.3.6 Game Environment Designer

The game environment designer is the viewpoint where users model a game environment in which a scenario takes place in. This viewpoint is designed based on the game environment concept described in Section 5.3.5.1. It uses WYSIWYG visualisation which allows users to position game objects, proximity trigger and checkpoints to construct the environment in which the game-play will set in and virtual camera visually on a 2D space (see Figure C.22). It also uses dynamic option interface to aid users to reference to game objects which has been defined in the game object designer viewpoint.

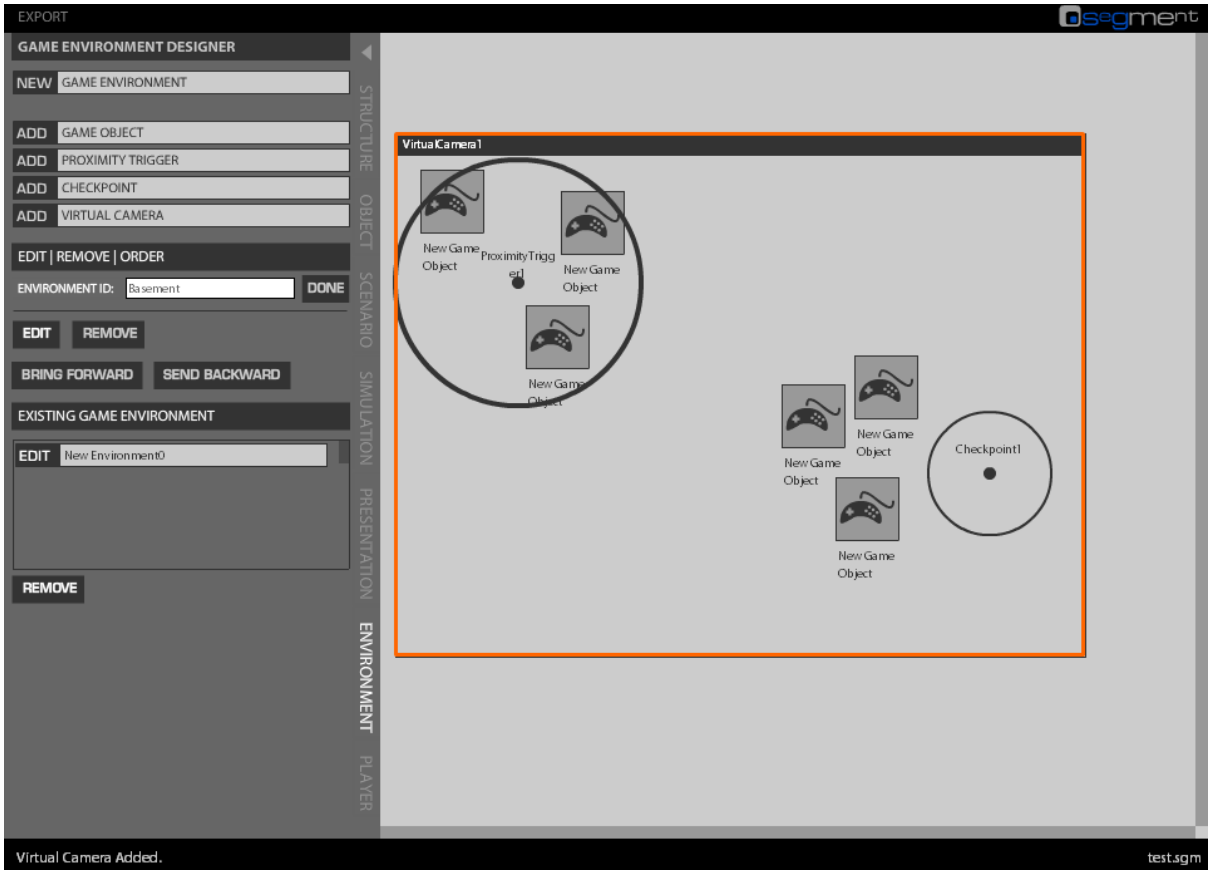

Figure C.22: Modelling game environment in Game Environment Designer.

C.2.3.7 Game Player Designer

In the game player designer, domain expert can specify the player's avatar, the inventory size which limits the storage of virtual items, the game attribute associated, the data to be tracked and the mapping of game controls to game object's action. This viewpoint is designed based on the game player concept described in Section 5.3.9. It uses the guided data entry interface and dynamic option interface to aid users in specifying all the details required to define attributes associated to game player (see Figure C.23).

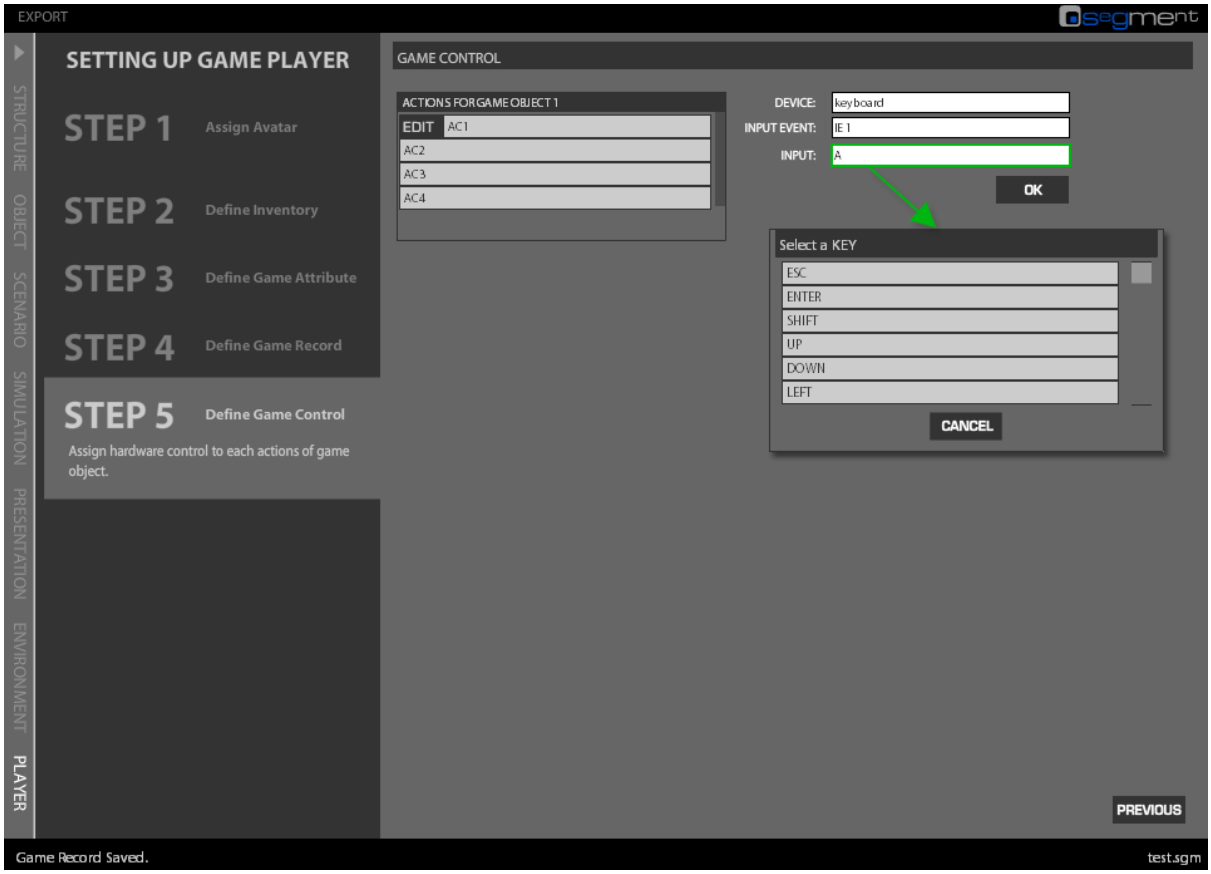

Figure C.23: Modelling game player in Game Player Designer.

C.2.4 Generation of Game Content Model

Our SeGMEnt tool provides the interface aid for domain expert to document the design of a serious game that is compliant to the Game Content Model. Underlying the UI is the data which needs to be saved, processed and exported into eXtensible Markup Language (XML) format (see Section C.3 for more details) which can be transformed by our MDE tools which we have chosen. In our implementation, we traverse through all the data structures stored in the shell module at _level0 of the Adobe Flash movie stack (refer to Section C.6.2.1) that represent all the concepts of Game Content Model and mark it with XML tags in the XML structure illustrated in Figure C.24. On the highest level, the serious game element consist of basic description of the game such as author and title, the game player concept, the game structure concept and collections of reusable concepts such as game presentation, game simulation, game object and game scenario. These reusable concepts are referred to using a unique reference identifier in the game structure such as gamePresentationID, gameSimulationID and gameScenarioID.

```
<!--Definition for Serious Game-->
  <!--Game Player-->
  <!--Game Presentations-->
  <!--Game Simulations-->
  <!--Game Scenarios-->
  <!--Game Objects-->

  <!--Game Structure-->
     <!--Definition for Game Structure-->
         <!--Definition for Game Context-->
         <!--PedagogicEvent-->

         <!--EventTrigger-->
         <!--Reference to Game Presentation-->
     <!--Definition for Game Context-->
         <!--PedagogicEvent-->
         <!--EventTrigger-->
         <!--Reference to Game Simulation-->
         <!--Reference to Game Scenario-->

…
```

Figure C.24: XML Structure for Game Content Model

In our implementation, each of the viewpoints store data in an array of objects as shown in Figure C.25.

```
//Data Structure for Game Structure
var gameStructure:Array = new Array();

//Data Structures for Game Scenarios
var stockGameScenarios:Array = new Array();
var gameScenario = undefined;
var currentGameScenario: Number = 0;

//Data Structure for Game Simulations
var stockGameSimulations:Array = new Array();
var gameSimulation = undefined;
var currentGameSimulation: Number = 0;

//Data Structure for Game Presentation
var stockGamePresentations:Array = new Array();
var gamePresentation = undefined;
var currentGamePresentation: Number = 0;

//Data Structure for Game Scenario
var stockGameScenarios:Array = new Array();
var stockGameActs:Array = new Array();
var currentGameScenario:Number = 0;

//Data Structure for Game Environment
var stockGameEnvironments:Array = new Array();
var gameEnvironment = undefined;
var currentGameEnvironment: Number = 0;

//Data Structure for Game Object
var stockGameObjects:Array = new Array();
var gameObject = undefined;
var currentGameObject:Number = 0;

//Data Structure for Game Player
var gamePlayer:Object = new Object();
var gameObjects:Array = new Array();
```

Figure C.25: ActionScript 2.0 snippet that shows the data structure for each viewpoint.

Each of the arrays is traversed in the order of the XML structure presented in Figure C.24 and every data in each object are accessed. XML tags are added to mark each of these data and a string of text in XML format is constructed as shown in Figure C.26. Once all the data have been tagged, the Game Content Model is sent to the GCMCreator.php to generate an XML file which will be processed by our MDE tool to transform into the subsequent model (Game Technology Model) in our model driven approach.


```
function exportToXML_fn(){
    var xml:String = "";
    xml += "<seriousgame>";

    //<!--Game Player-->
    xml += "<gameplayer>";

    //Avatar
    xml += "<avatar>";
    xml += "<assignmenttype>" + gamePlayer._avatar._assignment + "</assignmenttype>";
    if(gamePlayer._avatar._assignment == "fixed"){
        xml += "<id>" + gamePlayer._avatar._referenceID + "</id>";
        xml += "<name>" + gamePlayer._avatar._referenceName + "</name>";
    }
    xml += "</avatar>";

    //Inventory
    if(gamePlayer._inventory != undefined){
        xml += "<inventorysize>" + gamePlayer._inventory + "</inventorysize>";
    }

    //Game Attributes
    xml += "<gameattributes>";
    for(var i=0; i < gamePlayer._gameAttributes.length; i++){
        var xmlGameAttribute = gamePlayer._gameAttributes[i];
        xml += "<gameattribute>";
        xml += "<id>" + xmlGameAttribute._id + "</id>";
        xml += "<name>" + xmlGameAttribute._name + "</name>";
        xml += "<startvalue>" + xmlGameAttribute._startValue + "</startvalue>";
        xml += "<endvalue>" + xmlGameAttribute._endValue + "</endvalue>";
        xml += "</gameattribute>";
    }
    delete xmlGameAttribute;
    xml += "</gameattributes>";

    //Game ...
    xml += "</gamestructure>";
    xml += "</seriousgame>";

    var lv:LoadVars = new LoadVars();

    lv.GCM = xml;

    lv.send("GCMCreator.php","_blank", "POST");
}
```

Figure C.26: ActionScript 2.0 snippet that export Game Content Model to XML format.

C.3 Model Representation

In our model-driven approach, the data that describe the serious game model documented via the modelling environment should be represented a format which can hold the aspects of serious game defined in the Game Content Model as tokens of information. This information token must be able to represent a set of attributes with its values as a whole.

In our framework, we favour the use of open data format such as eXtensible Markup Language (XML). XML is a specification for defining how information is stored (Bray, Paoli, Sperberg-McQueen, Maler, & Yergeau, 2008) like any other earlier form of initiatives such as the CASE Data Interchange Format (CDIF) (Flatscher, 2002) and Portable Common Tool Environment (PCTE) (ECMA, 1997) introduced for use to store software engineering data. It offers great flexibility for defining the data format for representing models. In addition, XML can easily accept additional information from the automated transformation process between the models for MDE. Furthermore it is also well supported by MDE technologies such as Eclipse Modelling Framework (EMF) and Generic Modelling Environment (GME) (Ledeczi, et al., 2001) making it the ideal choice for representing data-model. This makes XML is a viable option for defining data-model in our model driven framework.

C.4 Model Transformations

In our model driven serious game development framework, the Game Content Model is generated by our software tool, SeGMEnt (Serious Games Modelling Environment) and represented in XML format. The Game Content Model then undergoes a transformation to be translated into the Game Technology Model, a computational model independent of platform, using a MDE tool. The MDE tool can be developed using existing MDE technologies such as EMF and GME as described in Section 3.7 or implemented using any programming languages with XML parser facility. In our model-driven serious games development framework, we have developed a custom transformation tool in PHP so to allow data to be processed on the server-side. The transformation of Game Content Model to Game Technology is mainly a process of refining data and reformatting it into a computation independent model by reorganisation of data into programmatic structures. This also involves the addition of programmatic statement calls to the relevant Game Technology Model component's function to process the relevant data. Whereas the transformation from Game Technology Model to Game

Software Model further refines the data by adding in platform specific data to the source model (Game Technology Model).

There are two approaches to have a Game Content Model translated to a Game Technology Model. The first approach is to interpret the source model (Game Content Model) and then weave the additional information into the source model in order to produce the target model (Game Content Model). This approach would require the source model to be well structured and the translation process is just merely locating the token of information and weaving in the additional information into the source model to create the target model. The second approach is to traverse through the entire source model to locate the required token of information and a new target model is composed by structurally reformatting data in the source model and adding in the additional information. In our case, we opted for second approach because it does not constraint us to the structure of the source model.

The process of model translation, as described earlier, requires (1) traversing the XML document structure in search for the marked elements which will be (2) reformatted and may include more information to construct the new model or artefacts. In PHP, there are various approaches for traversing XML document structure and this includes the Simple XML (core library in PHP 5.0), XML Expat parser and XML DOM (Document Object Model). In our prototype, we decided to use the Simple XML approach mainly because it is a simpler approach to traverse a document structure. This validates the notion that creation of a proprietary model translator and a code generator can be simple. In our approach, we specify the path of the XML structure to locate the marked data. Once marked data has been located, data can be accessed and re-marked with additional information. Whenever there is more than one child node in the structure, the *foreach()* construct is used to iterate over the tree structure as shown in Figure C.27.

```
<%php
//Loading XML doc
$xmlDoc = simplexml_load_file("seriousgame.xml");

////More implementations… (omitted in this example)

//Game Player
$GTM_XML = $GTM_XML . "<gamePlayer>";
$GTM_XML = $GTM_XML . "<attributes>";
$gameAttribute = $xmlDoc->gameplayer->gameattributes->gameattribute;
foreach($gameAttribute as $ga)
{
   $GTM_XML = $GTM_XML . "<" . $ga->name ."StartValue.setValue>";
   $GTM_XML = $GTM_XML . "<constant value=\"" . $ga->startvalue . "\"/>";
   $GTM_XML = $GTM_XML . "<" . $ga->name ."StartValue.setValue/>";

   $GTM_XML = $GTM_XML . "<" . $ga->name ."EndValue.setValue>";
   $GTM_XML = $GTM_XML . "<constant value=\"" . $ga->endvalue . "\"/>";
   $GTM_XML = $GTM_XML . "<" . $ga->name ."EndValue.setValue/>";
}

$GTM_XML = $GTM_XML . "</attributes>";
$GTM_XML = $GTM_XML . "</gamePlayer>";

////More implementations… (omitted in this example)
%>
```

Figure C.27: Snippet of code from the Game Technology Model translator to locate a marked data and iterating through the tree structure.

The new model or artefact is created by transforming data into a refined new format of data through the process of string concatenation. By using the string object, we can easily add in additional information and format data to the desired format. An example of Game Content Model generated from SeGMEnt is shown in Figure C.28 and the Game Technology Model which is transformed using our proprietary tool is shown in Figure C.29. The Game Technology Model will later be transformed into Game Software Model before it is used for generation of artefacts which can either be software code or a compiled software artefact.

The transformation of Game Technology Model to Game Software Model in our prototype will require inclusion of platform specific components such as windows management, file system, timer, graphics wrapper and physics wrapper to the Game Technology Model. This is less complicated compared to the transformation of Game Content Model to Game Technology Model because this transformation will only require the additional data to be appended to the existing Game Technology Model. This additional data will only include XML tags that mark the interfaces of the components to be included to enable code pairing during artefact generation process.

```
<gameObject>
   <id>fireman</id>
   <objectAttributes>
      <mass>75.0</mass>
      <solid>true</solid>
      <vitalAttribute>
         <id>health</id> <startValue>100.0</startValue>
<endValue>0.0</endValue>
      </vitalAttribute>
      <vitalAttribute>
         <id>energy</id> <startValue>100.0</startValue>
<endValue>0.0</endValue>
      </vitalAttribute>
   </objectAttributes>
   <objectAppearance>
      <spriteSource>asset/fireman/sprite.png</spriteSource>
      <spriteDimension>
        <height>20</height>
        <width>20</width>
      </spriteDimension>
   </objectAppearance>
   <objectAction>
      <id>idle</id>
      <animation>
         <startFrame>1</startFrame> <endFrame>3</endFrame>
      </animation>
   </objectAction>
   <objectAction>
      <id>walkLeft</id>
      <motion>
         <forceValue>5.0</forceValue> <forceAngle>-180</forceAngle>
      </motion>
      <animation>
         <startFrame>4</startFrame> <endFrame>8</endFrame>
      </animation>
      <vitalUpdate>
         <attributeID>energy</attributeID>
         <arithmeticOperator>-</arithmeticOperator>
         <constant>0.1</constant>
      </vitalUpdate>
   </objectAction>
</gameObject>
```

Figure C.28: XML describing the Fireman game object generated from SeGMEnt

```
<!--Element for Game Object-->
<gameObject id="fireman" type="actor">
  <attributes>
    <collidable value="true"/>
    <mass value="75.0"/>
    <inventory capacity="0.0"/>
    <appearance.value>
      <assetManager.load id="assetManager">
        <constant value="asset/fireman/sprite.png"/>
      </assetManager.load>
    </appearance.value>
    <animations>
      <spriteSequence id="idle">
        <frame top="0" bottom="20" left="0" right="20"/>
        <frame top="0" bottom="20" left="20" right="40"/>
        <frame top="0" bottom="20" left="40" right="60"/>
      </spriteSequence>
      <spriteSequence id="walkLeft">
        <frame top="0" bottom="20" left="60" right="80"/>
        <frame top="0" bottom="20" left="80" right="100"/>
        <frame top="0" bottom="20" left="100" right="120"/>
        <frame top="0" bottom="20" left="120" right="140"/>
        <frame top="0" bottom="20" left="140" right="160"/>
      </spriteSequence>
    </animations>
  </attributes>
  <operations>
    <gameObject.update>
      <if>
        <condition>
          <gameObject.actionState/>
          <operator value="=="/>
          <constant value="idle"/>
        </condition>
        <animationManager.setSequence>
          <gameObject id="fireman"/>
          <constant value="idle"/>
        </animationManager.setSequence>
      </if>
      <elseif>
        <condition>
          <gameObject.actionState/>
          <operator value="=="/>
          <constant value="walk"/>
        </condition>
        <physicManager.applyForce id="physicManager">
          <gameObject id="fireman"/>
          <constant value="0.5"/>
          <constant value="180"/>
        </physicManager.applyForce>
        <animationManager.setSequence>
          <gameObject id="fireman"/>
          <constant value="walkLeft"/>
        </animationManager.setSequence>
      </elseif>
      <animationManager.update id="animationManager">
        <gameObject id="fireman"/>
      </animationManager.update>
    </gameObject.update>
    <gameObject.render>
      <renderManager.render id="renderManager">
        <gameObject.appearance id="fireman"/>
      </renderManager.render>
    </gameObject.render>
    <gameObject.cleanup></gameObject.cleanup>
  </operations>
</gameObject>
```

Figure C.29: XML definition of the Game Object in Game Technology Model. Elements in bold are translated token of information from Game Content Model which has been reorganised into programmable format.

C.5 Code Generation

Our basic code generation tool is also implemented using PHP 5.0. It uses a similar approach to the approach described in our Game Content Model to Game Technology Model transformation tool (see Section C.4). Each of the marked tokens is located and then translated into ActionScript 2.0 syntax. Each of the token of information either map to a single line of code or a segment of codes defined in the code template. The final code is built up based on the structure of Game Software Model and generated as textual artefacts ready to be copied to clipboard and used in Adobe Flash authoring environment.

C.6 Summary

In this appendix section, we have implemented our custom-built modelling environment and MDE tools to show the applicability of our model-driven game development framework. Our serious game modelling environment (SeGMEnt) in Adobe Flash is the UI module to our model driven game development framework (refer to Part (1) in Figure 5.3). The SeGMEnt encapsulates all the concepts in our Game Content Model described in Section 5.3 using the UIs components we described in Section C.2.2. In our SeGMEnt, we have chosen a mix of UIs specifically designed to aid non-technical domain experts in modelling aspects of serious game. We have developed our own algorithm to create an extended state-like notation (flow visualisation), assistive user interfaces (dynamic option interface) and statement construction interface. In addition, we have also designed and implemented other helpful interfaces such as guided data entry interface and the WYSIWYG visualisation. These user interfaces are used wisely to encapsulate all the concepts of Game Content Model within the seven design viewpoints. Implementing these highly customised user interfaces components in Adobe Flash is not without its technical challenges. Despite being the most popular platform for developing rich experience application for the web, it has its own constraints in representing and managing graphical object (movie clip), architecture of project, support for 2D drawing and object referencing are some to name. Many of our solutions have to be designed around the limitation of the platform and these are described in Section C.2.2.

In our model driven framework, the models are represented in XML to make marking and locating of marked information easier, and therefore simplifying the task of model transformation. Framework developers will need to have a good understanding of each of the model (Game Content Model, Game Technology Model and Game Software Model) before they can implement refinement to each model.

In our framework, we choose not to be constrained by the structure of models and have opted to implement our MDE tool that locates the marked information in the source model and rebuilds the target model from scratch. Development of the MDE transformation tool is proven to be much simpler and straight forward especially with modern XML programming interfaces such as Simple XML in PHP. We have also chosen to implement our own code generator tool and developed our own code mapping technique to pair the XML tags (information token) with segment of codes.

GAME BIBLIOGRAPHY

Dance Dance Revolution, 1998, Konami, Konami

Darfur is Dying, 2006, Susana Ruiz & Take Action games, mtvU

Grand Theft Auto IV, Rockstar North, Take-two Interactive

Guitar Hero, 2005, Harmonix, RedOctane

Halo: Reach, 2010, Bungie, Microsoft Game Studios

Killzone 3, 2011, Guerrilla Games, Sony Computer Entertainment

Need for Speed: Shift, 2009, Slightly Mad Studios, Electronic Arts

Pacman, 1980, Namco, Midway Games

Pro Evolution Soccer 2011, 2010, Konami Computer Entertainment Tokyo, Konami

Red Dead Redemption, 2010, Rockstar San Diego, Rockstar

Resident Evil 5, 2009, Capcom, Capcom

Starcraft II: Wings of Liberty, 2010, Blizzard Entertainment, Blizzard Entertainment

Unreal Tournament III, 2007, Epic Games, Midway Games

Warcraft III: Reign of Chaos, 2002, Blizzard Entertainment, Blizzard Entertainment

Worm Fort: Under Siege, 2004, Team 17, Sega

REFERENCES

Adams, E. (2004). Designer's Notebook: Designing with Gameplay Modes and Flowboards Retrieved 31 October, 2009, from http://www.gamasutra.com/view/feature/2101/designers_notebook_designing_.php

Adams, E., & Rollings, A. (2006). *Fundamental of Game Design*. US: Prentice Hall.

Agrawal, A., Karsai, G., & Ledeczi, A. (2003). *An end-to-end domain-driven software development framework.* Paper presented at the Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA, USA.

Aguilera, M. d., & Mendiz, A. (2003). Video games and education: (education in the face of a "parallel school"). *Computers in Entertainment (CIE), 1,* 10-10.

Altunbay, D., Cetinkaya, E., & Metin, M. G. (2009, 20th May). *Model Driven Development of Board Games* Paper presented at the First Turkish Symposium on Model-Driven Software Development (TMODELS), Bilkent University, Ankara Turkey.

Ang, C. S., & Rao, G. S. V. R. K. (2004). Designing Interactivity in Computer Games a UML Approach. *International Journal of Intelligent Games and Simulation, 3*(2), 62-69.

Baker, A., Navarro, E. O., & Hoek, A. v. d. (2005). An experimental card game for teaching software engineering processes. *Journal of Systems and Software, 75*(1-2), 3-16.

Basin, D., Doser, J., & Lodderstedt, T. (2006). Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol., 15*(1), 39-91. doi: http://doi.acm.org/10.1145/1125808.1125810

Bates, B. (2004). *Game Design* (2 ed.). Boston, MA: Thomson Course Technology PTR.

BECTa. (2001). Computer Games in Education: Aspects Retrieved 19 February, 2008, from http://partners.becta.org.uk/index.php?section=rh&rid=13588

BECTa. (2006). Computer Games in Education: Findings Report Retrieved 19 February, 2008, from http://partners.becta.org.uk/index.php?section=rh&rid=13595

Bethke, E. (2003). *Game Development and Production*. Plano, Texas: Wordware Publishing.

Bézivin, J. (2004). In Search of a Basic Principle for Model Driven Engineering. *UPGRADE, V*(2), 21-24.

Bézivin, J., & Gerbé, O. (2001). *Towards a Precise Definition of the OMG/MDA Framework*. Paper presented at the 16th IEEE international conference on Automated software engineering, Coronado Island, San Diego, CA, USA.

Bishop, L., Eberly, D., Whitted, T., Finch, M., & Shantz, M. (1998). Designing a PC Game Engine. *IEEE Computer Graphics and Applications, 18*(1), 46-53. doi: 10.1109/38.637270

Björk, S., & Holopainen, J. (2004). *Patterns in Game Design*: Charles River Media.

Björk, S., Lundgren, S., & Holopainen, J. (2003). *Game Design Patterns*. Paper presented at the Level Up, University of Utrecht, The Nethelands.

Bogost, I. (2007). Persuasive Games: How I Stopped Worrying About Gamers And Started Loving People Who Play Games Retrieved 20 February, 2008, from http://www.gamasutra.com/view/feature/1543/persuasive_games_how_i_stopped_.php

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (2008, 5th December 2009). Extensible Markup Language (XML) 1.0 (Fifth Edition), from http://www.w3.org/TR/2008/REC-xml-20081126/

Burnside, J. B. (2008). Determining fire hazards when educators decorate their classroom in Clinton, Missisippi.: National Fire Academy.

Carter, C., Rhalibi, A. E., Merabti, M., & Bendiab, A. T. (2010). *Hybrid Client-Server, Peer-to-Peer Framework for MMOG*. Paper presented at the IEEE International Conference on Multimedia and Expo (ICME), Suntec City.

Carter, C., Rhalibi, A. E., Merabti, M., & Price, M. (2009). *Homura and Net-Homura: The Creation and Web-based Deployment of Cross-Platform 3D Games.* Paper presented at the International

Workshop on Ubiquitous Multimedia Systems and Applications (UMSA'09), St. Petersburg, Russia.
Ceri, S., Daniel, F., Facca, F. M., & Matera, M. (2007). Model-driven Engineering of Active Context-awareness. *World Wide Web, 10*(4), 387-413.
Checkland, P., & Scholes, J. (1990). *Soft Systems Methodology in Action*. West Sussex, England: John Wiley & Son.
Choi, S., Jang, J.-w., Mohanty, S., & Prasanna, V. K. (2003). Domain-Specific Modeling for Rapid Energy Estimation of Reconfigurable Architectures. *The Journal of Supercomputing, 26*(3), 259-281.
Colt, H., Crawford, S., & Galbraith, O. (2001). Virtual Reality Bronchoscopy Simulation*: A Revolution in Procedural Training. *CHEST, 120*(4), 1333-1339.
Connolly, T. M., & Stansfield, M. (2007). From eLearning to Games-Based eLearning. *International Journal of Information Technology and Management, 26*(2/3/4), 188-208.
Cook, E., & Hazelwood, A. (2002). An active learning strategy for the classroom-"who wants to win … some mini chips ahoy? *Journal of Accounting Education, 20*(4), 297-306.
Cook, S., Jones, G., Kent, S., & Wills, A. C. (2007). *Domain-Specific Development with Visual Studio DSL Tools*. Crawfordsville, Indiana: Addison Wesley.
Crawford, C. (1982). *The Art of Computer Game Game Design*.
Darken, R., McDowell, P., & Johnson, E. (2005). Projects in VR: the Delta3D open source game engine. *Computer Graphics and Applications, IEEE, 25*(3), 10-12.
ECMA. (1997). Portable Common Tool Environment (PCTE) - C Programming Language Binding.
El-Rhalibi, A., Hanneghan, M., Tang, S., & England, D. (2005). *Extending Soft Models to Game Design: Flow, Challenges and Conflicts*. Paper presented at the DiGRA 2005 - the Digital Games Research Association's 2nd International Conference, Simon Fraser University, Burnaby, BC, Canada.
Espejo, R. (1990). The Viable System Model. *Systemic Practice and Action Research, 3*(3), 219-221.
Everett, J. (2003). Building a business simulation for kids: the making of Disney's hot shot business *Computers in Entertainment (CIE), 1,* 18-18.
Fall, A., & Fall, J. (2001). A domain-specific language for models of landscape dynamics. *Ecological Modelling, 141*(1-3), 1-18.
FAS. (2006a). Harnessing the power of video games for learning, Summit on Educational Games 2006, from http://fas.org/gamesummit/Resources/Summit%20on%20Educational%20Games.pdf
FAS. (2006b). Harnessing the power of video games for learning, Summit on Educational Games 2006.
Favre, J.-M., & Nguyen, T. (2005). Towards a Megamodel to Model Software Evolution Through Transformations. *Electronic Notes in Theoretical Computer Science, 127*(3), 59-74.
Flatscher, R. G. (2002). Metamodeling in EIA/CDIF---meta-metamodel and metamodels. *ACM Trans. Model. Comput. Simul., 12*(4), 322-342. doi: http://doi.acm.org/10.1145/643120.643124
Fondement, F., & Silaghi, R. (2004). *Defining Model Driven Engineering Processes*. Paper presented at the 3rd Workshop in Software Model Engineering (WiSME2004), Lisbon, Portugal.
France, R., Ghosh, S., Dinh-Trong, T., & Solberg, A. (2006). Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer, 39*(2), 59-66.
France, R., & Rumpe, B. (2007). *Model-driven Development of Complex Software: A Research Roadmap*. Paper presented at the 29th International Conference of Software Engineering: Future of Software Engineering, Minneapolis, MN.
Fullerton, T. (2008). *Game Design Workshop, Second Edition: A Playcentric Approach to Creating Innovative Games* (2 ed.). Burlington, MA: Morgan Kaufmann.
Furtado, A. W. B. (2006). *SharpLudus: Improving Game Development Experience Through Software Factories And Domain-Specific Languages.* Masters, Federal University of Pernambuco, Pernambuco, Brazil.

Furtado, A. W. B., & Santos, A. L. M. (2006a). *Defining and Using Ontologies as Input for Game Software Factories*. Paper presented at the 5th Brazilian Symposium on Computer Games and Digital Entertainment, Recife Antigo, Brazil.

Furtado, A. W. B., & Santos, A. L. M. (2006b). *Using Domain-Specific Modeling towards Computer Games Development Industrialization*. Paper presented at the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon USA.

Gagne, R. M. (1970). *The Conditions of Learning and Theory of Instruction* (2nd ed.). New York: Holt, Rinehart & Winston.

Gal, V., Prado, C. L., Natkin, S., & Vega., L. (2002). *Writing for video games*. Paper presented at the VRIC 2002, Laval France.

Gaševic, D., Djuric, D., & Devedžic, V. (2006). *Model Driven Architecture and Ontology Development*. Berling, Germany: Springer-Verlag.

Gregory, J. (2009). *Game Engine Architecture*. Natick, Massachusetts: A K Peters, Ltd.

Grønmo, R., Skogan, D., Solheim, I., & Oldevik, J. (2004). *Model-driven Web Services Development*. Paper presented at the 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'04), Taipei, Taiwan.

Hammond, J. L. (2007). Extending Frameworks with Domain-Specific Modelling Language. *Embedded System Engineering, 15,* 16-18.

Harel, D., & Kugler, H. (2001). The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). *Lecture Notes in Computer Science, Integration of Software Specification Techniques for Application in Engineering*(3147), 325 - 354.

Hemme-Unger, K., Flor, T., & Vogler, G. (2003). *Model driven architecture development approach for pervasive computing*. Paper presented at the Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA, USA.

Heskett, J. (2005). *Design: A Very Short Introduction*. New York: Oxford University Press.

Hildenbrand, T., & Korthaus, A. (2004). *A Model-Driven Approach to Business Software Engineering*. Paper presented at the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 2004), Orlando, Florida.

Hill, J., Tambe, S., & Gokhale, A. (2007). *Model-driven Engineering for Development-time QoS Validation of Component-based Software Systems*. Paper presented at the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), Tucson, Arizona.

Hoffman, H. G., Garcia-Palacios, A., Carlin, C., Furness, T. A. I., & Botella-Arbona, C. (2003). Interfaces that heal: Coupling real and virtual objects to cure spider phobia. *International Journal of Human-Computer Interaction, 15*, 469-486.

Hunicke, R., LeBlanc, M., & Zubek, R. (2004). *MDA: A Formal Approach to Game Design and Game Research.* Paper presented at the Challenges in Game AI Workshop, AAAI04.

Irvine, C. E., Thompson, M. F., & Allen, K. (2005). CyberCIEGE: Gaming for Information Assurance. *IEEE Security and Privacy, 3*(3), 61-64.

Järvinen, A. (2007). *Introducing Applied Ludology: Hands-on Methods for Game Studies*. Paper presented at the Digra 2007 Situated Play: International Conference of the Digital Games Research Association, Tokyo, Japan.

Javed, A. Z., Strooper, P. A., & Watson, G. N. (2007). *Automated Generation of Test Cases Using Model-Driven Architecture*. Paper presented at the Proceedings of the Second International Workshop on Automation of Software Test.

Jenkins, H., Klopfer, E., Squire, K., & Tan, P. (2003, October 2003). Entering The Education Arcade. *Computers in Entertainment (CIE), 1,* 17-17.

Katara, M., Kervinen, A., Maunumaa, M., Paakkonen, T., & Satama, M. (2006). *Towards Deploying Model-Based Testing with a Domain-Specific Modeling Approach.* Paper presented at the

Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings.
Kelleher, C. (2006). *Motivating Programming: Using storytelling to make computer programming attractive to middle school girls.* PhD Technical Report, Carnegie Mellon University.
Kelleher, C., Cosgrove, D., Culyba, D., Forlines, C., Pratt, J., & Pausch, R. (2002). *Alice2: Programming without Syntax Errors* Paper presented at the User Interface Software and Technology, Paris, France.
Kelly, H., Howell, K., Glinert, E., Holding, L., Swain, C., Burrowbridge, A., & Roper, M. (2007). How to build serious games. *Communications of the ACM, 50*(7), 44-49.
Kelly, S., & Tolvanen, J. P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. Hoboken, New Jersey: Wiley-IEEE Computer Society Press.
Kent, S. (2002). *Model Driven Engineering*. Paper presented at the 3rd International Conference of Integrated Formal Methods (IFM2002), Turku, Finland.
Kienzle, J., Denault, A., & Vangheluwe, H. (2007). Model-Based Design of Computer-Controlled Game Character Behavior In G. Engels, B. Opdyke, D. C. Schmidt & F. Weil (Eds.), *Lecture Notes in Computer Science* (Vol. 4735/2007, pp. 650-665): Springer Berlin / Heidelberg.
Kim, C., & Agrawala, A. K. (1989). Analysis of the Fork-Join Queue. *IEEE Trans. Comput., 38*(2), 250-255. doi: 10.1109/12.16501
Kleppe, A. G., Warmer, J. B., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture : Practice and Promise*: Addison-Wesley.
Koper, R., & Manderveld, J. (2004). Educational modelling language: modelling reusable, interoperable, rich and personalised units of learning. *British Journal of Educational Technology, 35*(4), 537-551.
Koster, R. (2004). *A Theory of Fun for Game Design*: Paraglyph.
Ledeczi, A., Maroti, M., Bakay, A., Karsa, G., Garrett, J., Thomason, C., . . . Volgyesi, P. (2001). *The Generic Modeling Environment.* Paper presented at the Workshop on Intelligent Signal Processing, Budapest, Hungary.
Leont'ev, A. N. (1977). *Activity and Consciousness* (N. Schmolze & A. Blunden, Trans.): Progress Publishers.
Lodderstedt, T., Basin, D., & Doser, J. (2002). *SecureUML: A UML-Based Modeling Language for Model-Driven Security*. Paper presented at the Model Engineering, Concepts, and Tools 5th International Conference, Dresden, Germany.
Malone, T. W., & Lepper, M. R. (1987). Making Learning Fun: A Taxonomy of Intrinsic Motivations for Learning. In R. E. Snow & M. J. Farr (Eds.), *Aptitude, Learning and Instruction* (Vol. 3): Lawrence Erlbaum Associates.
Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., & Resnick, M. (2004). *Scratch: A Sneak Preview*. Paper presented at the Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing.
Manuel, R. (2002). A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing, 1*, 74-83.
Miller, J., & Mukerji, J. (2003). MDA Guide Vesion 1.0.1 (Vol. 28th December): OMG.
Natkin, S., Vega, L., & Grünvogel, S. M. (2004). *A New Methodology for Spatiotemporal Game Design*. Paper presented at the International Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004), Reading, UK.
Nelson, M. J., & Mateas, M. (2007). *Towards Automated Game Design.* Paper presented at the 10th Congress of the Italian Association for Artificial Intelligence on AI*IA: Artificial Intelligence and Human-Oriented Computing, Rome, Italy.
Nielsen, L. (2013). *Personas - User Focused Design*: Springer.
Obrenovic, Z., Starcevic, D., & Selic, B. (2004). A Model-Driven Approach to Content Repurposing. *IEEE Multimedia, 11*(1), 62-71. doi: http://dx.doi.org/10.1109/MMUL.2004.1261109

OMG. (2001, 21 December 2008). Model Driven Architecture (MDA), from http://www.omg.org/docs/ormsc/01-07-01.pdf
OMG. (2003). MDA Guide Vesion 1.0.1 Retrieved 2008, 28th December, from www.omg.org/docs/omg/03-06-01.pdf
Onder, B. (2002). Writing Adventure Game. In F. o. D. Laramée (Ed.), *Game Design Perspectives* (pp. 28-43). Hingham, Massachusetts: Charles River Media.
Pearce, C. (2006). Productive Play: Game Culture From the Bottom Up. *Games and Culture, 1*(1), 17-24.
Pivec, M., & Dziabenko, O. (2004). Game-Based Learning in Universities and Lifelong Learning: "UniGame: Social Skills and Knowledge Training" Game Concept. *Journal of Universal Computer Science, 10*(1), 14-26.
Poole, J. D. (2001, 18th - 22nd June). *Model-Driven Architecture: Vision, Standards and Emerging Technologies.* Paper presented at the 15th European Conference on Object-Oriented Programming (ECOOP2110), Workshop on Adaptive Object-Models and Budapest, Hungary.
Prensky, M. (2001). *Digital Game-Based Learning*: Paragon House.
Reyno, E. M., & Cubel, J. Á. C. (2008). *Model-Driven Game Development: 2D Platform Game Prototyping*. Paper presented at the GAMEON' 2008, Valencia, Spain.
Rollings, A., & Adams, E. (2003). *Andrew Rollings and Ernest Adams on Game Design*: New Riders Publishing.
Rollings, A., & Morris, D. (2004). *Game Architecture and Design: A New Edition*. Indianapolis, Indiana: New Riders.
Rouse, R. (2001). *Game Design: Theory & Practice* (2 ed.). Plano, Texas: Wordware Publishing.
Roussou, M. (2004). Learning by doing and learning through play: an exploration of interactivity in virtual environments for children. *Computers in Entertainment (CIE) 2,* 10-10.
Salen, K., & Zimmerman, E. (2003). *Rules of Play: Game Design Fundamentals* The MIT Press.
Sarinho, V., & Apolinário, A. (2008). *A Feature Model Proposal for Computer Games Design*. Paper presented at the VII Brazilian Symposium on Computer Games and Digital Entertainment, Belo Horizonte.
Sarinho, V., & Apolinário, A. (2009). *A Generative Programming Approach for Game Development.* Paper presented at the VII Brazilian Symposium on COmputer Games and Digital Entertainment, Rio de Janeiro, Brazil.
Sawyer, B., & Smith, P. (2008). Serious Games Taxonomy Retrieved 26 March, 2008, from http://www.dmill.com/presentations/serious-games-taxonomy-2008.pdf
Schauerhuber, A., Wimmer, M., & Kapsammer, E. (2006). *Bridging existing Web modeling languages to model-driven engineering: a metamodel for WebML*. Paper presented at the Second international workshop on model driven web engineering (MDWE'06), Palo Alto, California.
Schmidt, D. C. (2006). Guest Editor's Introduction: Model-Driven Engineering. *Computer, 39*(2), 25-21.
Seidewitz, E. (2003). What models mean? *Software IEEE, 20*(5), 26-32.
Shetty, S., Nordstrom, S., Ahuja, S., Di, Y., Bapty, T., & Neema, S. (2005). *Systems integration of large scale autonomic systems using multiple domain specific modeling languages.* Paper presented at the Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the.
Sottet, J.-S., Calvary, G., Favre, J.-M., Coutaz, J., Demeure, A., & Balme, L. (2006). *Towards Model Driven Engineering of Plastic User Interfaces* Paper presented at the MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica.
Squire, K., Barnett, M., Grant, J. M., & Higginbotham, T. (2004). *Electromagnetism Supercharged! Learning Physics with Digital Simulation Games.* Paper presented at the 6th International Conference on Learning Sciences, Santa Monica, California.

Tang, S., & Hanneghan, M. (2005). *Educational Games Design: Model and Guidelines*. Paper presented at the 3rd International Game Design and Technology Workshop (GDTW'05), Liverpool, UK.

Tang, S., & Hanneghan, M. (2008). *Towards a Domain Specific Modelling Language for Serious Game Design*. Paper presented at the 6th International Game Design and Technology Workshop (GDTW'08), Liverpool, UK.

Tang, S., & Hanneghan, M. (2010a). Designing Educational Games: A Pedagogical Approach. In P. Zemliansky & D. Wilcox (Eds.), *Design and Implementation of Educational Games: Theoretical and Practical Perspectives* (pp. 108-125). Hershey, PA: IGI Global.

Tang, S., & Hanneghan, M. (2010b). *A Model-Driven Framework to Support Development of Serious Games for Game-based Learning*. Paper presented at the 3rd International Conference on Developments in e-Systems Engineering (DESE2010), London, UK.

Tang, S., & Hanneghan, M. (2011a, 5-6 December). *Fusing Games Technology and Pedagogy for Games-Based Learning Through a Model Driven Approach.* Paper presented at the Proceedings of IEEE Colloquium on Humanities, Science & Engineering Research (CHUSER 2011), Penang, Malaysia.

Tang, S., & Hanneghan, M. (2011b, 6-8 December). *Game Content Model: An Ontology for Documenting Serious Game Design.* Paper presented at the Proceedings of 4th International Conference on Developments in e-Systems Engineering (DESE2011), Dubai, UAE.

Tang, S., Hanneghan, M., & Carter, C. (2012, 4-5 October). *A Platform Independent Model for Model Driven Serious Games Development.* Paper presented at the 6th European Conference on Games Based Learning (ECGBL 2012), Cork, Ireland.

Tang, S., Hanneghan, M., & Carter, C. (2013). A Platform Independent Game Technology Model for Model Driven Serious Games Development. *The Electronic Journal of e-Learning, 11*(1), 61-79.

Tang, S., Hanneghan, M., & El-Rhalibi, A. (2004, November 25-27). *Designing Challenges and Conflicts: A Tool for Structured Idea Formulation in Computer Games.* Paper presented at the 5th International Conference on Intelligent Games and Simulation (GAME-ON 2004), Het Pand, Ghent, Belgium.

Tang, S., Hanneghan, M., & El-Rhalibi, A. (2006). *Modelling Dynamic Virtual Communities within Computer Games: A Viable System Modelling (VSM) Approach,* . Paper presented at the 4th International Game Design and Technology Workshop (GDTW'06), Liverpool, UK.

Tang, S., Hanneghan, M., & El-Rhalibi, A. (2007). *Pedagogy Elements, Components and Structures for Serious Games Authoring Environment*. Paper presented at the 5th International Game Design and Technology Workshop (GDTW 2007), Liverpool, UK.

Tang, S., Hanneghan, M., & Rhalibi, A. E. (2009). Introduction to Games-Based Learning. In T. M. Connolly, M. H. Stansfield & L. Boyle (Eds.), *Games-Based Learning Advancements for Multi-Sensory Human Computer Interfaces: Techniques and Effective Practices* (pp. 1-17). Hershey: Idea-Group Publishing.

Taylor, M. J., Baskett, M., Hughes, G. D., & Wade, S. J. (2007). Using Soft Systems Methodology for Computer Game Design. *Systems Research and Behavioral Science, 24*, 359-368.

Taylor, M. J., Gresty, D., & Baskett, M. (2006). Computer game-flow design. *Computers in Entertainment (CIE), 4*(1), Article No. 5.

Telfer, R. (1993). *Aviation Instruction and Training*. Aldershot: Ashgate.

Thramboulidis, K. C. (2004). *Using UML in Control and Automation: A Model Driven Approach*. Paper presented at the 2nd IEEE International Conference on Industrial Informatics INDIN'04, Berling, Germany.

Tolvanen, J.-P. (2006). Domain-Specific Modeling: How to Start Defining Your Own Language. Retrieved from http://www.devx.com/enterprise/Article/30550

Veit, M., & Herrmann, S. (2003). *Model-view-controller and object teams: a perfect match of paradigms*. Paper presented at the Proceedings of the 2nd international conference on Aspect-oriented software development, Boston, Massachusetts.

Wada, H., & Suzuki, J. (2006). Modeling Turnpike: A Model-Driven Framework for Domain-Specific Software Development *Satellite Events at the MoDELS 2005 Conference* (pp. 357-358).

WFPFoodForce. (2008) Retrieved 20 February, 2008, from http://www.food-force.com

Zagal, J. P., Mateas, M., Fernández-Vara, C., Hochhalter, B., & Lichti, N. (2005). *Towards an Ontological Language for Game Analysis.* Paper presented at the DiGRA 2005 - the Digital Games Research Association's 2nd International Conference, Selected Papers, Simon Fraser University, Burnaby, BC, Canada.

Zyda, M. (2005). From Visual Simulation to Virtual Reality to Games. *Computer, 38,* 25-32.