VIDEO GAME DEVELOPMENT ONTOLOGY

Glauco Ofranti Trindade

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Mario Roberto Folhadela Benevides
Ivan José Varzinczak

Rio de Janeiro
Dezembro de 2015

VIDEO GAME DEVELOPMENT ONTOLOGY

Glauco Ofranti Trindade

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

____

Prof. Mario Roberto Folhadela Benevides, Ph.D.

____

Prof. Ivan José Varzinczak, Ph.D.

____

Prof. Alexandre Rademaker, D.Sc.

____

Profa. Sheila Regina Murgel Veloso, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

DEZEMBRO DE 2015

Trindade, Glauco Ofranti

Video Game Development Ontology / Glauco Ofranti Trindade. – Rio de Janeiro: UFRJ/COPPE, 2015.

XIV, 200 p.: il.; 29,7 cm.

Orientadores: Mario Roberto Folhadela Benevides

Ivan José Varzinczak

Dissertação (mestrado) – UFRJ/ COPPE/ Programa de Engenharia de Sistemas e Computação, 2015.

Referencias Bibliográficas: p. 169-176.

1. Ontologias. 2. Video Games. 3. Desenvolvimento de Video Games. I. Benevides, Mario Roberto Folhadela *et al*. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III Título.

To my family.

AKNOWLEDGEMENTS

This dissertation was the hardest and harshest challenge I have ever taken in my life and many people have, directly or indirectly, helped me in developing it over these three years. Here, I would like to take the opportunity to thank them.

First of all, I would like to thank Geraldo Bonorino Xexéo for the short time where he initially supervised this dissertation and acted as my advisor. I thank him for the knowledge imparted that helped me in writing this dissertation.

I would like to thank my advisors Mario Benevides and Ivan Varzinczak. I thank them for giving me the opportunity to conclude this dissertation, the invaluable advice that helped me conclude this dissertation and the confidence they gave me.

I would like to thank Alexandre Rademaker and Sheila R. Murgel Veloso which were members of the examining board. I thank them for their invaluable contributions that helped me improve this dissertation.

I would like to thank my friend Filipe Braida. I thank him for insightful advice regarding dissertations, giving me a model of dissertation to make this one, the rides he gave me to UFRJ and to home, but more importantly I thank for his friendship.

I would like to thank of the many colleagues I made in COPPE post-graduate program. I thank them for exchanging of ideas, sharing the difficulties and teaming up to overcome many of the program challenges.

I would like to thank my hypnotherapist Heloísa Helena. I thank her for helping me overcome my internal struggles, keeping my mind healthy and developing the necessary mental strength to conclude the dissertation. Thanks to you I have a more positive outlook of my life.

Finally, I would like to thank my family for their constant support and strength to conclude this dissertation. I am grateful to be blessed to have such wonderful parents because I would not be here if not for their efforts. Thank you very much for everything you have done for me.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

ONTOLOGIA DE DESENVOLVIMENTO DE VÍDEO GAMES

Glauco Ofranti Trindade

Dezembro/2015

Orientadores: Mario Roberto Folhadela Benevides
Ivan José Varzinczak

Programa: Engenharia de Sistemas e Computação

O desenvolvimento de vídeo games herda muitos dos problemas encontrados em projetos de engenharia de software como complexidade e escopo inviável. Entretanto, ele é uma atividade de desenvolvimento de software mais complexa quando comparada a outras porque envolve equipes multidisciplinares altamente especializadas compostas de programadores, designers, escritores, artistas, etc., que causa um déficit de comunicação entre essas equipes. Além disso, não existem padrões para documentação e vocabulário na literatura e indústria de vídeo games. Neste contexto, ontologias podem ser uma potencial solução porque elas fornecem uma representação consensual e compartilhada do conhecimento de um domínio que pode ser usada para solucionar problemas de comunicação e auxiliar em atividades de desenvolvimento e engenharia de software. Portanto, o objetivo deste trabalho é construir a Ontologia de Desenvolvimento de Vídeo Games que tem o objetivo de facilitar a identificação de requisitos técnicos no game design. A ontologia é construída seguindo uma metodologia de construção composta de cinco fases: aquisição de conhecimento, especificação, conceptualização, implementação e avaliação. A ontologia é validada usando seus termos e relações para modelar uma parte do gameplay de um vídeo game. O resultado deste trabalho é uma ontologia testada e validada tecnicamente e implementada em OWL 2.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

VIDEO GAME DEVELOPMENT ONTOLOGY

Glauco Ofranti Trindade

December/2015

Advisors: Mario Roberto Folhadela Benevides

Ivan José Varzinczak

Department: Computer Science Engineering

Video game development inherits many of the problems found in software engineering projects such as project complexity and unrealistic scope. However, it is a more complex software development activity when compared to others because it involves highly specialized multidisciplinary teams composed of programmers, designers, scriptwriters, artists, etc. which causes a communication gap between those teams. Also, there are no standards for documentation and vocabulary in the video game literature and the industry. In this context, ontologies can be a potential solution because they provide a consensual and shared representation of the knowledge of a domain which can be used to solve communication problems and assist in software engineering and development activities. Thus, this work has the objective of building the Video Game Development Ontology which has the objective of facilitating the identification of technical requirements in the game design. The ontology is built following a building methodology composed of five phases: knowledge acquisition, specification, conceptualization, implementation and evaluation. The ontology is validated by using its terms and relations to model a gameplay segment of a video game. The result of this work is a technically tested and validated ontology implemented in OWL 2.

INDEX

Chapter 1 – Introduction 1
1.1 – Motivation 1
1.1.1 – Creativity and Design Problems 2
1.1.2 – Software Application Problems 4
1.1.3 – Software Engineering Problems 4
1.1.4 – Multidisciplinary Problems 6
1.1.5 – Documentation Problems 6
1.1.6 – Requirements Engineering Problems 8
1.1.7 – Solution 9
1.2 – Objectives 12
1.3 – Hypothesis 13
1.4 – Organization 13
Chapter 2 – Ontologies 14
2.1 – Knowledge 14
2.2 – History of Ontologies 16
2.3 – Definitions of Ontology in Computer Science 18
2.4 – Ontologies as Knowledge Artifacts 20
2.5 – Applications of Ontologies 23
2.5.1 – Ontologies in Knowledge Engineering 25
2.5.2 – Ontologies in Computer Science 26
2.5.3 – Ontologies in Software Engineering 27
2.6 – What Are Ontologies Made Of? 28
2.7 – Differences between Ontologies and Models 31
2.8 – Types of Ontologies 34
2.9 – Ontological Engineering 35
2.10 – Ontology Design Principles 36
2.11 – Ontology Development Methodologies 38
2.12 – Ontology Development Tools 41
2.12.1 – Ontology Representation Languages 41
2.12.2 – Ontology Development Environments 43
Chapter 3 – Games 45
3.1 – What Are Games? 45
3.1.1 – What Are Games Made Of? 48
3.2 – What Are Video Games? 50

3.2.1 – What Are Video Games Made Of? 53
3.3 – Game Design 57
3.4 – Video Game Development Process 59
3.4.1 – Video Game Development Phases 60
3.4.2 – Video Game Development Roles 62
3.4.3 – Video Game Development Documents 66
Chapter 4 – Related Work 70
4.1 – Vocabularies for Game Design 70
4.2 – Informal Video Game Knowledge Models 73
4.3 – Formal Video Game Knowledge Models 76
4.4 – Game Ontologies 77
4.5 – Conclusion 80
Chapter 5 – Building Methodology 82
5.1 – METHONTOLOGY 82
5.1.1 – Ontology Development Process 82
5.1.2 – Ontology Life Cycle 84
5.2 – Ontology Building Activities 86
5.2.1 – Knowledge Acquisition 86
5.2.2 – Specification 86
5.2.3 – Conceptualization 88
5.2.4 – Formalization and Implementation 89
5.2.5 – Evaluation 90
Chapter 6 – Specification 92
6.1 – Purpose 92
6.2 – Intended Users 93
6.3 – Characteristics 93
6.4 – Formality 94
6.5 – Knowledge Sources 94
6.6 – Scope 95
Chapter 7 – Conceptualization of Internal Modules 101
7.1 – Game Object Module 101
7.2 – Attribute Module 104
7.2.1 – Atomic Attribute Types 107
7.3 – Event Module 107
7.4 – Action Module 110

7.5 – State Module 112
7.6 – Space Module 115
7.6.1 – Discrete Space 117
7.6.2 – Continuous Space 119
7.6.3 – Spatial Attributes, Actions, States and Events 121
7.7 – Time Module 121
7.7.1 – Discrete and Continuous Time 124
7.7.2 – Rewinding Time 125
Chapter 8 – Conceptualization of External Modules 126
8.1 – External Object Module 126
8.2 – Hardware Module 127
8.3 – Software Module 128
8.4 – Player Module 129
8.5 – Input Module 130
8.5.1 – Non-physical Input 132
8.5.2 – Physical Input 132
8.6 – Output Module 133
8.6.1 – Non-physical Output 135
8.6.2 – Physical Output 135
8.6.3 – Audio Output Actions 136
8.7 – Video Output Module 136
8.8 – Asset Module 139
8.9 – Video Game Module 140
Chapter 9 – Implementation 143
9.1 – Implementation Process Overview 143
9.2 – Implementation Phase Steps 144
9.3 – Implementation Process Phases 145
Chapter 10 – Evaluation 146
10.1 – Evaluation Process Overview 146
10.2 – Ontology Verification Findings 147
10.3 – Ontology Validation 152
10.3.1 – Identification of Game Elements 152
10.3.2 – Ontology Extensions 158
10.3.3 – Modelling the Gameplay Segment 159
10.3.4 – Problems in the Modelling Activity 160

10.3.5 – Ontology Validation Conclusion 161
Chapter 11 – Conclusion 162
11.1 – Final Considerations 162
11.2 – Comparison to Other Game Ontologies 163
11.3 – Limitations 164
11.4 – Contributions 165
11.5 – Future Work 166
Bibliographic References 169
Appendix A – OWL 2 177
A.1 – What is OWL? 177
A.2 – OWL Types 178
A.3 – OWL Features 178
A.3.1 – Basic Elements 178
A.3.2 – Equality and Inequality 179
A.3.3 – Property Characteristics 180
A.3.4 – Property Restrictions 180
A.3.5 – Complex Classes 182
A.4 – OWL 2 New Features 183
A.4.1 – Property Chains 183
A.5 – OWL 1 and OWL 2 Limitations 183
A.5.1 – Difference between Classes and Individuals 184
A.5.2 – Expressivity Limits 184
Bibliographic References 184
Appendix B – Conceptualization Tables 186
Appendix C – Competency Questions 199
C.1 – Internal Module Questions 199
C.2 – External Module Questions 200

LIST OF FIGURES

Figure 1 – Mario jumping 10
Figure 2 – Runtime game engine architecture (GREGORY, 2014) 55
Figure 3 – Decomposition of a game within a development team (LEWIS *et al.*, 2007) 63
Figure 4 – Ontology Development Process (CORCHO *et al.*,2005) 83
Figure 5 – Ontology life cycle (CORCHO *et al.*, 2005) 84
Figure 6 – Game Object Module 104
Figure 7 – Attribute Module 106
Figure 8 – Event Module 110
Figure 9 – Action Module 111
Figure 10 – *Monster Hunter 4 Ultimate*, *Great Sword* Move Chart 113
Figure 11 – State Module 115
Figure 12 – Space Module Hierarchy 117
Figure 13 – Space Module Relations 118
Figure 14 – Time Module Hierarchy 123
Figure 15 – Time Module Relations 123
Figure 16 – External Object Module Hierarchy 126
Figure 17 – External Object Module Relations 127
Figure 18 – Hardware Module 128
Figure 19 – Software Module 129
Figure 20 – Player Module 130
Figure 21 – Input Module 131
Figure 22 – Output Module 134
Figure 23 – Video Output Module 137
Figure 24 – Asset Module 139
Figure 25 – Video Game Module 141
Figure 26 – First stage of *Super Mario bros.* 152

LIST OF TABLES

Table 1 – List of some existing ontologies 24
Table 2 – Identified Visual Assets 155
Table 3 – Identified Audio Assets 156
Table 4 – Player Inputs 156
Table 5 – Space Logic 157
Table 6 – Specific logic 157
Table 7 – Game Object module taxonomy 187
Table 8 – Game Object module relations 187
Table 9 – Game Object module axioms 188
Table 10 – Attribute module taxonomy 188
Table 11 – Attribute Module relations 189
Table 12 – Attribute module axioms 189
Table 13 – Event module taxonomy 189
Table 14 – Event module relations 190
Table 15 – Event module axioms 190
Table 16 – Action module taxonomy 190
Table 17 – Action module relations 191
Table 18 – Action module axioms 191
Table 19 – State module taxonomy 191
Table 20 – State module relations 192
Table 21 – State module axioms 192
Table 22 – Space module relations 193
Table 23 – Space module axioms 193
Table 24 – Time module relations 193
Table 25 – Time module axioms 193
Table 26 – External Object module taxonomy 194
Table 27 – External Object module relations 194
Table 28 – Hardware module relations 194
Table 29 – Hardware module axioms 194
Table 30 – Software module relations 195
Table 31 – Software module axioms 195
Table 32 – Player module relations 195

Table 33 – Player module axioms 195
Table 34 – Input module taxonomy 196
Table 35 – Input module relations 196
Table 36 – Input module axioms 196
Table 37 – Output module taxonomy 196
Table 38 – Output module relations 197
Table 39 – Output module axioms 197
Table 40 – Video Output module axioms 197
Table 41 – Asset module taxonomy 197
Table 42 – Asset module relations 198
Table 43 – Asset module axioms 198
Table 44 – Video Game module relations 198
Table 45 – VGDO data properties 198

Chapter 1 – Introduction

1.1 – Motivation

Video games are an important part of society culture nowadays. The Entertainment Software Association (ESA) "2015 Essential Facts About the Computer and Video Game Industry" report shows that video games are a strong engine for economic growth. They have evolved into a mass medium: more than 150 million Americans play video games and 42 percent play video games regularly, or at least three hours per week. In 2014, the industry sold over 135 million games and generated more than $22 billion in revenue. Fifty two percent of total game sales were generated by purchases of digital content, including online subscriptions, downloadable content, mobile applications, and social networking games (ESA, 2015).

The evolution of video games into a mass medium can be credited at how they have quickly evolved technically and artistically these last decades. From simple pixel screens that did not provide detail to photo realistic games that provide physics simulations and cinematic experiences that engrosses the players into the medium, video games can provide experiences not available in movies and books. However, such quick evolution came with high costs. Compared to the first generation of video games where a development team was composed roughly of five people and had a development time of half a year, they have evolved to be large projects employing hundreds of people and development time measured in years (KANODE & HADDAD, 2009).

The end products of other creative industries like fashion, music, and movies are unchangeable after the release or production, but games are similar to conventional software products that can evolve incrementally with updates (KASURINEN *et al*,. 2014) making it an evolving product by nature. Because of its nature, a "perfect" project scope will never be achieved, but it is the goal of the manager to develop a solid scope that will help guide the project to its conclusion (KANODE & HADDAD, 2009). Thus, to achieve this objective there is the video game development process.

Video game development is an iterative and non-linear process (Flynt and Salem, 2004) since many features are introduced, modified or eliminated during the process. It can be

divided in three phases: pre-production, production and post-production. Pre-production consists in activities such as design, prototyping features of the game, feasibility study and requirements identification. Generally, the end product of this phase is a design document that will be used as reference to transform the design in functional software. The production phase mainly consists in the production of software code, integration of assets (images, videos) into the software and quality assurance (correction of bugs). Post-production involves marketing and maintenance (patches that correct bugs that passed the QA process) of the game (SALAZAR *et al.*, 2012).

Video game development is characterized by a high level of creativity when compared to other fields of software development. That is because video games cover a multitude of themes and genres, and represent a heterogeneous group of different products with varying requirements and business goals (KASURINEN *et al.*, 2014) and because they count on highly specialized multidisciplinary teams, having simultaneously software developers, designers, musicians, scriptwriters, artists and many other professionals (PETRILLO *et al.*, 2008) depending on the size of the project. This makes video games different from most other software application domains, since its development presents unique challenges originated from the multiple disciplines involved (KANODE & HADDAD, 2009). Thus, video game projects bring with them a myriad of problems with them, each problem being rooted in their respective discipline.

I classify the problems according to the following characteristics of a video game: it is a software application; it is a software engineering project; it is made by a multidisciplinary team; and it is a creative endeavor. Also there are two specific problems regarding software engineering that deserve a more detailed view: documentation and requirements engineering. Each type of problem can cause problems of other types, which can have a cascading effect. I will first talk about problems of being a creative work.

1.1.1 – Creativity and Design Problems

A fundamental difference from other software is that game software aims to provide an experience instead of a function. Because of that the game requirements elaboration is much more complex, therefore subjective elements as the “fun” does not have efficient techniques for its determination (CALLELE *et al.*, 2005).

WINGET & SAMPSON (2011) assert that video game development is a process that is primarily concerned with design problems, as opposed to production. Design accounts for the majority of game development challenges because in many cases it cannot be fully solved or even anticipated at the outset of the development process since the interaction between game elements is unpredictable. Another problem is that the elaboration of the game design document (GDD) is generally made by professionals with little or no technical background, making this document informal and not accurate.

CALLELE et al. (2005) assert that it is difficult to assess the player experience early in the development cycle for significant progress must be made on building the underlying game engine infrastructure before gameplay testing can begin. This is a particularly high-risk scenario because of the likelihood that new requirements will emerge as gameplay testing continues, new requirements that must be tracked, and for which test plans must be developed. The emerging requirements may even force significant changes to the fundamental architecture of the system that, in extreme cases, may cause project failure. This occurred during the development of *Xenoblade X* developed by *Monolith Soft* and *Nintendo Software Planning & Development*. In an Iwata Asks[1] , the developers revealed that in the middle of development they decided to make *Xenoblade X* compatible with online play. According to them, this decision caused a "mass construction" (revision) in the game to change the main character into an avatar (customizable character with no defined personality) and rewrite some of the story to match with the content (NINTENDO, 2015).

Problems in the production of the game assets such as music, images, models, scripts, etc., are of creative nature since they are a labor of creativity. Failure at providing those assets according to schedule can slow down other teams since assets need to be integrated with the game code as well as they need to be evaluated by other organizations (for example, determination if the game content is adequate to an age group).

---

[1] Iwata Asks is a series of interviews conducted by the late Nintendo Global President Satoru Iwata with key creators behind the making of Nintendo games and hardware. Those interviews are available at http://iwataasks.nintendo.com.

1.1.2 – Software Application Problems

As a video game is a software application, the most common problems are from technical nature such as wrong implementation of features, bugs, optimization, etc. According to BLOW (2004), video games are hard. The hardest part of making a game has always been the engineering. In past times, game engineering was mainly about low-level optimization—writing code that would run quickly on the target computer, leveraging clever little tricks whenever possible. But in the past decade prior 2004, games have ballooned in complexity causing the primary technical challenge to be simply getting the code to work to produce an end result that bears some resemblance to the desired functionality.

The overall project size and complexity and the highly domain-specific requirements are the main difficulties that causes problems in the development of games (BLOW, 2004). Those requirements are related to the skill sets needed by the development team such as 3D mathematics, artificial intelligence, linear algebra, programming in a specific language, algorithms, specific hardware knowledge, etc. As a result of the increased technical complexity, game developers carry a lot of technical risks (determining accurately how a feature will interact with the rest of the system is impossible) as well as game design risks (how will this never-implemented feature feel to the end user?) (BLOW, 2004). One of the worst technical problems that can happen is a feature of the finished design of the game being impossible to be implemented with the available technology. It can cause a revision of the design of the game or a change of the actual technology being used, halting the progress of the development and damaging the schedule. This problem generates problems to members outside of the programming team.

1.1.3 – Software Engineering Problems

The design and engineering of video game software is a subset of another relatively young discipline, software engineering. This is why the video game industry inherits all problems that come naturally in a software engineering project. That is, the percentage of software delivered on time, within budget, and without faults is incredibly low (CONGDON, 2008). While software application problems are technical, problems from a software engineering project are related to management and planning of the project.

FLYNT & SALEM (2004) assert that the biggest reason of games project imperfection is the failure in clearly establishing the project scope. If a project does not have a well-established target, emergent requirements can cause significant structural changes in the system's architecture, causing serious problems (CALLELE *et al.*, 2005). The development teams lose themselves in the scope when some difficulties arise: problems with the exaggerated size and complexity of the project, in addition to facing highly specific requirements of the games domain (BLOW, 2004). However, the main cause of scope problems is the common situation where new functionalities are added during the development phase, increasing the project's size. This practice is known in the industry as feature creep.

Those assertions are supported by PETRILLO *et al.*'s (2008) survey that shows that all the main problems of traditional software industry are also found in the games industry, and it is possible to affirm that they are much related. In both contexts, for example, the unreal scope was pointed out as critical, as the problems with requirements definition. Also, FLOOD (2003) claims that all game development postmortems reveal the same problems: the project was delivered behind schedule; it contained many defects; the functionalities were not the ones that had originally been projected; a lot of pressure and an immense amount of development hours in order to complete the project. Those problems are clearly related to poor planning and management of the project such as the lack of a realistic estimate on the initial plan of development, making the team not capable of finding a deadline for the projects; and the production of inaccurate estimates of time needed to complete a task due to lack of historical data that should assist the perception of time needed to execute it, causing cumulative schedule delays (FLYNT & SALEM, 2004).

It also should be noted that, according to CALLELE *et al.* (2005), the software engineering process in video game development is not clearly understood, hindering the development of reliable practices and processes for this field. The main reason for that is that the electronic games industry, for its competitiveness and corporative way of working, generally turns difficult to access internal data from projects (PETRILLO *et al*, 2008) as there are significant monetary disincentives for game companies to talk to any outsider about their development theories, practices, or processes, particularly if that outsider is going to go talk to

other companies about the same processes. Also there are copyright and intellectual property problems (WINGET & SAMPSON, 2011).

1.1.4 – Multidisciplinary Problems

According to CALLELE *et al.* (2005), the multidisciplinarity of the development team causes an important problem, present in the game industry but not in the traditional software industry, which is the communication among the teams. This mixture, in spite of being positive in the sense of having a more creative work environment, seems to produce a true split on the team, dividing it into “the artists” and “the programmers”. This division, that basically does not exist in the traditional software industry, is the main source of important misunderstanding problems (FLYNT & SALEM, 2004), since both teams believe to communicate clearly when using their specific vocabularies to express their ideas (CALLELE *et al.*, 2005). According to CONGDON (2008), both art and programming teams express their own objectives in their own terms, in their own types of documentation. These are appropriate for laying out the needs of both teams, but do not express how art and programming issues relate to each other in the game world.

This issue is magnified as team and projects grow in size, the group needs to know what stage each individual is at as well as the state of the entire project. Video game development has a challenge in that information that needs to be expressed has to be presented to people of many disciplines. If the game content is not expressed properly to each discipline then the continuity of the design of the game world may get lost (CONGDON, 2008). All of those problems along with the unique nature of video games projects impact in the generation of documents, an important process in a software engineering project.

1.1.5 – Documentation Problems

Effective documentation is essential for quality control as it forces development team members to review their own. It also introduces the element of accountability to software production and is there to help make clear what has been done in the past. It is known that software engineering projects are often being poorly documented. This combination of high complexity and poor documentation is perfect for creating confusion and mistakes. These mistakes end up costing valuable time and money for projects that already have enormous budgets. While software engineers are trying to establish their own documentation standards,

game designers have not been quick to contribute to the solution. As such, documentation dealing specifically with video game design is scarce and not well developed. Part of the problem is how a large number of disciplines come together to produce games (CONGDON, 2008).

The multidisciplinarity of a game development team can be observed from the documents described by WINGET & SAMPSON (2011) interviews with game development team members. There are different kinds of design documents, like technical design documents, which precisely describe the game's technical challenges and proposed solutions; pitch and concept documents, which are often used for promotional and fundraising; and miscellaneous others such as impromptu sketches or whiteboard diagrams and charts. CONGDON (2008) claims that developing documentation for multiple disciplines requires an understanding of what each party needs to accomplish and what they use to accomplish it, in other words domain knowledge is needed. When documentation is developed it is important to be as useful as possible while being fast and easy to create so that team members are more likely to use it properly.

According to FULLERTON (2014), best practices for producing games are evolving to recognize the need for flexibility and iteration as part of the game development process. Many developers now use a mix of agile development methods and traditional software production methods to produce their games. The core difference between those distinct development methods is a focus on creating working software versus documentation and managing the team so that it can respond to discoveries in the process, rather than following a predetermined plan. Because of this shift in the game development process and its iterative nature is that GDDs are literally out of date the moment they are written (SCHELL, 2014). BETHKE (2003) comments that he has never seen a completed design document, and one of the reasons is that game design documents need to be maintained through the course of production. With time-to-market pressures so prevalent, it is easy to see how documentation maintenance is given low priority. WINGET & SAMPSON (2011) conclude in their article that traditional design documentation loses significant accuracy and descriptive ability as development progresses, and should be complemented by records that communicate more of the creative process in video game production and design. These materials are: the iterative

versions of game assets and the game itself as it may exist at any time, as well as models or other abstractions of the game.

Undoubtedly the GDDs are the most important documents in the game development process, as the game's overall design is most frequently communicated in these documents written in the pre-production phase. Its uses vary from studio to studio and project to project, but it always serves as a general reference point for members of the development team (WINGET & SAMPSON 2011). The GDDs are an important piece of the requirements gathering process. It is from those documents that the many type of requirements are identified such as number of assets needed, technology needed to implement the game design, professionals needed, etc. From those requirements other types of documentation are written such as technical documents describing the technology used to implement the game design.

1.1.6 – Requirements Engineering Problems

Requirements engineering for media production in video game development is particularly challenging because of the interactions between the requirements of the video game artifact, the requirements of the tools needed to create the video game artifact, and the strongly differentiated user groups (CALLELE *et al.*, 2005). According to SALAZAR *et al.* (2012), requirements engineering best practices may support preproduction and production stages, by bringing structure, detail and establishing relationships among video-game elements in order to improve the best experiences during the play time. According to KANODE & HADDAD (2009), gathering all the needed requirements will cut down on the number of iterations needed, mitigate the late addition of features (feature creep), reduce errors due to miscommunication with the customers (the designers) and identify unstated requirements.

The identification of requirements start once the first version of the game design is finished. CALLELE *et al.* (2005) claim that, in a sense, the GDD is the requirements document as defined by the preproduction team. However, the GDD leads to challenges in the accuracy and reliability of the identified requirements. CALLELE *et al.* (2005) assert that by its nature as a creative work, a game design document is replete with implied information because significant elements of the game design documentation are informal, often with substantial visual content. Identifying these implications requires careful analysis,

understanding the ramifications of such implications require significant domain knowledge. If one attempts to formalize this document, they must understand large portions of both the preproduction and production realms, in other words, domain knowledge is required.

According to CALLELE *et al.* (2005), performing and managing the transformation of the GDD into requirements is complex. Each of these documents requires a different writing style and a single individual may not have the requisite writing skills to author materials for all purposes. In addition, creating the requirements document or specification document often requires considerable a priori knowledge of the available technology so that the requirements can be presented in context. There is also a multiplicative effect: each successive document is larger than the prior document as the author(s) attempt to precisely capture the required information. The authors must manage multiple stakeholder viewpoints, synthesizing a common domain language, numerous nonfunctional requirements, and inconsistencies as the project evolves. There are many types of requirements that are placed on a game such as creative, functional, technical, fiscal, license and temporal requirements (BETHKE, 2003).

CALLELE *et al.*'s (2005) investigation of factors leading to success or failure in video game development suggests that many failures can be traced back to problems with the transition from preproduction to production. There are three problems: how to transform documentation from its preproduction form to a form that can be used as a basis for production; how to identify implied information in preproduction documents; and how to apply domain knowledge without hindering the creative process. They conclude that creating documentation to support the transition from game design document through formal requirements and specifications is difficult, requiring significant preproduction and production domain knowledge to perform successfully. A formal process to support this transition would likely increase the reliability of the process.

1.1.7 – Solution

CALLELE *et al.* (2005) assert that requirements engineering requires the creation of a common (domain) language (and implied world model) specific to the task at hand because of the diversity of a game development team. Once all stakeholders fully commit to the domain language, then a set of requirements that capture the stakeholders wants and necessities can be generated. A common language, ontology, or vision is often mentioned as the solution to

communications issues between disparate stakeholders. In other words, management of the knowledge of the various domains of the game development process is required.

There is a lot of implied knowledge in GDDs and its materials (NIESENHAUS & LOHMANN, 2009). CALLELE *et al.* (2005) assert that at least three levels of implication can be identified from the GDD: implications that can be derived directly from the materials presented; implications that can only be derived with the introduction of general knowledge of the domain; and implications that can only be derived with the introduction of implementation details such as the target architecture. I will use as an example Figure 1, which is artwork from the game *Mario vs Donkey Kong* developed by Nintendo for the *Game Boy Advance*:

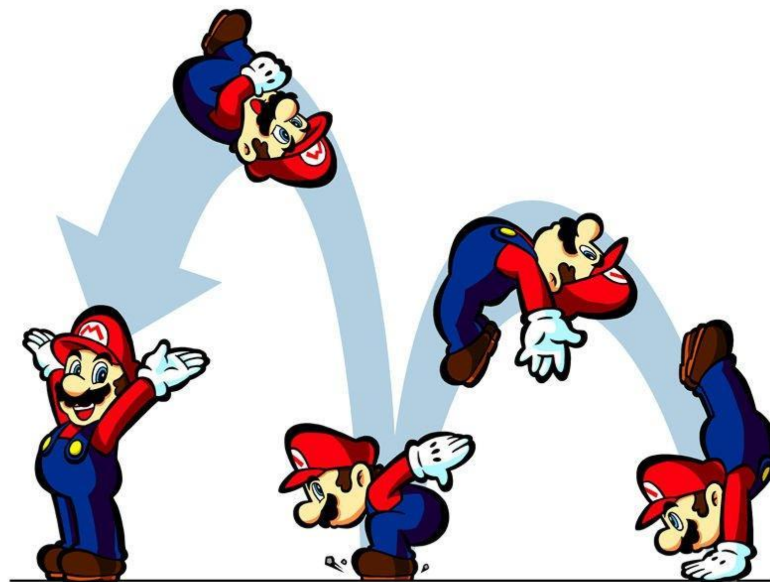

Figure 1 – Mario jumping

The first level of implication that can be derived from the image is that Mario can jump, how the player should see Mario, etc. In other words, what we see in the image should be seen in the game. The implications of the second level can be: an artist making an animation that represents the artwork, the game designer imposes restrictions to Mario jumps such as height; how the player input will make Mario execute the jump action; what sounds Mario makes when he jumps; etc. In other words, these implications are generally detected and handled by the creative part of the development team. Finally, the implications of the last level can be: since Mario can jump and land in a surface, it means that a collision and physics system must be implemented; how much time each jump animation plays; how many frames

each animation has; etc. In other words, these implications are handled by the technical part of the development team.

Therefore, a knowledge management solution should be adopted to solve these problems since in a single image many types of knowledge can be inferred and there are professionals of different knowledge domains. NIESENHAUS & LOHMANN (2009) propose that appropriate knowledge management solutions for video game development must satisfy a number of criteria:

- Adaptable: being easily adaptable to changing project demands such as the addition and removal of new features in a game;
- Lightweight: following the principles of simplicity and ease-of-use which means that the advantages that the solution brings are worth the effort and time needed to learn it;
- Immediate: adding immediate benefit to the project and all its participants such as being a knowledge repository that can easily be queried, modified and stored;
- Generic: providing a general solution for various projects which means that it can be used in many types of video games;
- Nonrestrictive: not dictating strict procedures but fostering creativity. It means that the ontology is simply a tool to be used by the development team.

Ontologies are promising candidates because they can satisfy those criteria as they are a form of structured knowledge of a domain, giving meaning to the objects and relations of the domain. They are adaptable because you can extend a generic domain ontology to describe a more specific domain within, they can be generic because there are generic ontologies that can be used in a number of different domains, they can be immediate because ontologies can be easily modified by software programs and can be used as tools to support other software, they can be lightweight because there is software that allows smooth browsing and editing of ontologies and they can be nonrestrictive because the users can set how rigid the ontology structure can be. Also, ontologies can improve communication between different disciplines because they are providing a shared understanding of the domain.

A game development team will surely benefit from ontologies that meet these criteria; the potential advantages they provide are many. As ontologies provide a shared and

consensual vocabulary, concepts and ideas from a game will have an unambiguous meaning and their relations will be explicitly exposed to all team members. The ontology will fill the communication gaps between the team members as they can communicate without misunderstandings since the team agreed to such vocabulary. Another advantage is the fact that it is a shared knowledge repository, team members can query the ontology for any doubts they have regarding certain concepts of the game in the event of a member being uncommunicable and his knowledge is required. Also, the knowledge repository will increase the speed of the development process because, according to SCHELL (2014), as real-world designer and developers work without a standardized vocabulary, whenever there is an ambiguity in some concepts the designer have to stop his work and explain what those concepts mean. Ontologies can be easily modified and support evolution mechanisms such as version control making them an appropriate form of documentation for the iterative and dynamic nature of the development process since the user will be able to see previous versions of the ontology enabling developers to see the rationale behind the game design changes. Finally, because of the formalism an ontology possess, automatic reasoning on them can be performed inferring implicit knowledge. This can help designers and other team members to find flaws in the game design, assets that needs to be made, patterns, etc. Even THORN (2013) asserts that a computer needs more formal, concrete, and explicit definitions to be given for all concepts that are to be found throughout the game. It demands that there is a systematic and precise linguistic means of breaking down reality conceptually to say what exists. A way of dividing up and cataloguing the world, in other words, an ontology.

Therefore, this work has the objective to introduce an ontology that provides a common vocabulary and assists in the transition between preproduction and production phases of the game development process by helping the identification implied knowledge in GDDs. Thus, it makes the gathering of requirements more accurate and reliable and, in consequence, mitigates several problems in the game development process. The ontology will be called the Video Game Development Ontology (VGDO) and it will be designed with the design and programming teams as the intended end-users.

1.2 – Objectives

The objectives of this work are:

- To propose an ontology that provides a common vocabulary and helps in the identification of technical requirements in the game design;
- To implement the ontology in an ontology representation language;
- To validate the ontology.

1.3 – Hypothesis

The hypothesis of this work is:

- It is possible to identify knowledge, from other domains present in the game development process, which is hidden implicitly in the game designer knowledge (images, videos, documents) using an ontology.

Thus, to arrive at an answer I need to make a new ontology and validate it.

1.4 – Organization

This dissertation is organized as such:

Chapters 2, 3 and 4 compose the theoretical basis of the work. Chapter 2 discusses the concepts of ontologies. Chapter 3 discusses video games. Those are discussed in great detail in order to understand what they are, what are they made of and how they are made. Chapter 4 presents related work regarding the proposal of knowledge structures to describe games and application of ontologies are discussed.

Chapter 5 presents the methodology of construction of the ontology.

Chapter 6 presents the specification of the ontology. Here the ontology purpose and scope are defined and the main terms to be used are identified.

Chapter 7 and 8 presents the conceptualization of the ontology.

Chapter 9 presents the implementation of the ontology. Here the ontology structure is formalized and implemented in an ontology representation language.

Chapter 10 presents the evaluation of the ontology. Here the ontology is validated following an evaluation process.

Finally, in the conclusion the potential many uses of the ontology in the game development process are discussed as well as the limitation of the work.

Chapter 2 – Ontologies

In this chapter, I will talk about ontologies and the relevant concepts surrounding it. First, I will talk about the importance of knowledge nowadays since an ontology is a formal description of shared knowledge in a domain. Second, I will talk about its evolution through history to provide the context in which it is used in Computer Science. Third, ontologies characteristics and potential applications will be presented. Finally, ontological engineering will be detailed and explained.

2.1 – Knowledge

"*Knowledge is power*". This is a famous aphorism found in the *Meditations Sacrae* (1597) written by Sir Francis Bacon, one of the founders of empiricism. It has great significance because the fact that many of humanity technological advancements were achieved because of the culmination of knowledge developed through the centuries and new knowledge derived or reasoned from it. Knowledge is so important and valuable nowadays that we have laws that protect intellectual property, patents that grant exclusivity rights to a certain idea or invention and corporations making their own knowledge secret. Depending on the domain of the subject, a person who has extensive knowledge of the domain can be worth millions and is highly desired in any kind of organization.

Companies realize more and more that the knowledge they possess (also known as corporate memory) is of essential importance for successful operation on the market. Such knowledge should be accessible for the appropriate people and should be maintained to be always up-to-date (STUDER *et al.*, 1998). Therefore, knowledge is now part of the capital and resources of organizations, and an efficient information system is a vital asset. In the kingdom of information, knowledge is king (GANDON, 2010). One of the definitions of knowledge is that it is the understanding of a subject area. It includes concepts and facts about that subject area, as well as relations among them and mechanisms for how they are combined to solve problems in that area. (GAŠEVIC *et al.*, 2009a)

The problem with knowledge is that its "power" is not easily attainable since it is intangible thus making it hard to represent formally. First, not all knowledge can be represented in a written or electronic form, because some knowledge is abstract and

subjective making it hard to capture its semantics or the domain of the subject dos not have an adequate vocabulary to communicate the knowledge.

Second, it is hard to acquire knowledge from knowledge sources. People might not be able to understand and learn the knowledge already available in books or the Internet because of many reasons: the vocabulary may cause ambiguity, how the knowledge is structured or how the knowledge is communicated to the reader. The knowledge can be perceived as incoherent and incomprehensible. Relevant knowledge items can appear in a multitude of different document formats: text documents, spreadsheets, presentation slides, database entries, Web pages, construction drawings, or email, to name but a few. The challenge lies in how you handle the knowledge (STAAB *et al.*, 2001).

Finally, many people mistake possessing information as the same thing as possessing knowledge. Raw information in large quantities does not by itself solve business problems, produce value, or provide competitive advantage. Information is useless without understanding of how to apply it effectively. But with the volume of information available increasing rapidly, turning information into useful knowledge has become a major problem (FENSEL, 2003).

The field of Knowledge Engineering (KE) exists to solve those issues regarding knowledge. KE is a broad research domain where the core issues include knowledge acquisition and the modeling of knowledge. Modeling knowledge consists in representing it in order to store it, to communicate it or to externally manipulate it (GANDON, 2010). The term KE is often associated with the development of expert-systems, involving methodologies as well as knowledge representation techniques (AHMED, 2008) and the people who perform them are knowledge engineers.

In KE, a number of activities were developed to be used in the development of practical knowledge bases: acquisition of human knowledge (from human experts or from other sources), understanding it properly, transformation into a form suitable for applying various knowledge representation formalisms, encoding it in the knowledge base using appropriate representation techniques, languages, and tools, verification and validation of knowledge by running the practical intelligent system that relies on it, and maintenance of the knowledge over time (GAŠEVIC *et al.*, 2009a).

Knowledge representation, one of the core issues of KE, raises the problem of the choice of a representation formalism that allows us to capture the semantics at play in the targeted pieces of knowledge. One approach that emerged in the late 80s is based on the concept of ontologies (GANDON, 2010). It has appeared with the aim of sharing and reusing knowledge in KE (JAZIRI & GARGOURI, 2010).

However, before talking about the role of ontologies in various domains of applications in more detail, it is necessary to understand the origin of this concept and the history behind its adoption in KE and Computer Science (CS).

2.2 – History of Ontologies

The history of ontologies in philosophy and CS are discussed in the works of GANDON (2010), CORCHO *et al.* (2006), JAZIRI & GARGOURI (2010), USCHOLD & TATE (1998) and especially in great detail in Guizzardi thesis (2005, Chapter 3). In this section, I will provide a brief history of ontologies based on the content of the mentioned works.

The term "ontology" was constructed from the Greek *Ontos* ("what is", "what exists") and *Logos* ("the discourse, "the study"). In philosophy, ontology is a fundamental branch of metaphysics, concerned with the concept of existence, the basic categories of existing and the most general properties of being. As a branch of philosophy, Ontology is the metaphysical study of the nature and relations of existence (GANDON, 2010, GUIZZARDI, 2005).

The ancient Greeks were concerned with the question: "what is the essence of things through the changes?" Many different answers to this question were proposed by Greek philosophers, from Parmenides of Elea, the precursor of ontology, to Aristotle, author of Metaphysics (CORCHO *et al.*, 2006).

In his study of the essence of things, Aristotle distinguished different modes of being to establish a system of categories (substance, quality, quantity, relation, action, passion, place and time) to classify anything that may be predicated (said) about anything in the world. The categorization proposed by Aristotle was widely accepted until the eighteenth century (CORCHO *et al.*, 2006).

In the modern age, Immanuel Kant (1724-1804) asserted that the essence of things is determined not only by the things themselves, but also by the contribution of whoever perceives and understands them. According to Kant, a key question is “what structures does our mind use to capture the reality?” The answer to this question leads to Kant’s categorization. Kant’s framework is organized into four classes, each of which presents a triadic pattern: quantity (unity, plurality, totality), quality (reality, negation, limitation), relation (inherence, causality, community) and modality (possibility, existence, necessity) (CORCHO *et al.*, 2006). A classification of categories, such as the ones mentioned above, is known as an ontology by philosophers (GUARINO, 1998).

The notion and the software artifact that we now name “ontologies” existed in computer science far before the term “ontology” was imported from philosophy. Back in the 70s, the notion of ontology was already used without being named as such and under various names in different knowledge representation frameworks of symbolic artificial intelligence. Even the relational schema of a database is a kind of ontological knowledge (GANDON, 2010).

In the beginning of the 1990s, ontologies have become a popular research topic investigated by several Artificial Intelligence (AI) research communities, including KE, natural-language processing and knowledge representation (STUDER *et al.*, 1998). Although ontology as a science comes from philosophy, it has mainly been developed by the AI community. This community has focused on developing reasoning mechanisms that would alleviate the task of enriching an ontology by addition of new concepts (JAZIRI & GARGOURI, 2010). Therefore, one can say that CS ontologies are children of AI that recently came to maturity and powerful conceptual tools of knowledge modeling (GANDON, 2010).

As time passed, the notion of ontology also became widespread in fields such as intelligent information integration, information retrieval on the Internet, and knowledge management. This is also caused many definitions of ontology to be proposed and evolved over time in those different fields (JAZIRI & GARGOURI, 2010). The reason for ontologies being so popular is in large part because what they promise: a shared and common understanding of some domain that can be communicated across people and computers. The main motivation behind ontologies is that they allow for sharing and reuse of knowledge

bodies in computational form (STUDER *et al.*, 1998). At this point, it is very important to take into account that 'an ontology' is not the same as 'ontology'. An ontology is a classification of categories, whereas ontology is a branch of philosophy (CORCHO *et al.*, 2006).

According to USCHOLD & TATE (1998), there was a highly varied and inconsistent usage of a wide variety of terms, most notably, "ontology", rendering cross-discipline communication difficult. However, this issue was mitigated because subsequent workshops that addressed various aspects of the field, including theoretical issues, methodologies for building ontologies, as well as specific applications in government and industry. This caused progress to be made toward understanding the commonality among the disciplines.

As mentioned above, in the field of CS there are several definitions for the term 'ontology'. It is important to understand the different aspects of ontologies that each field of CS use in order to define what an ontology really is, as well as to understand the common characteristics that those different interpretations have. Therefore, in the next section, I will explore some of the most important definitions that ontologies have in CS in order to answer the question "What is an ontology in CS?"

2.3 – Definitions of Ontology in Computer Science

According to GANDON (2010), the delay to reach a precise definition for a definition of the term "ontology" is probably largely because of the very abstract nature of the concept of ontologies. Once used outside philosophy, several interpretations of the concept of ontology are possible. The term ontology has an unquestionable definite meaning in the science of philosophy. Once imported into other domains, it loses some of its characteristics and gains others without any of these phenomena to be explained and defined by the borrowers (JAZIRI & GARGOURI, 2010).

In CS, one of the first definitions of ontology was "*an ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary*" (NECHES *et al.*, 1991).

A few years later, GRUBER (1995) propose a new definition: "*An ontology is an explicit specification of a conceptualization*". A conceptualization is viewed as an *abstract,*

simplified view of the world to be formally represented (GRUBER, 1995). This definition became the most quoted in literature and by the ontology community.

BORST (1997) has given an elaboration of GRUBER's definition, as follows: "*Ontologies are defined as formal specification of a shared conceptualization*". A year later, GRUBER's and BORST's definitions have been merged and explained by STUDER *et al.* (1998): "*Ontologies are explicit formal specification of a shared conceptualization*". According to the same authors, *conceptualization* refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. *Explicit* means that the type of concepts used, and the constraints on their use are explicitly defined. *Formal* refers to the fact that the ontology should be machine readable, which excludes natural language. *Shared* reflects the notion that an ontology captures consensual knowledge, that is, it is not private for some individuals, but accepted by a group.

In AI, the term "ontology" has largely come to mean one of two related things (CHANDRASEKARAN *et al.*, 1999): *a representation vocabulary, often specialized to some domain or subject matter or a body of knowledge describing some particular domain*. In both cases, there is always an associated underlying data structure that represents the ontology (GAŠEVIC *et al.*, 2009b).

The artificial intelligence interpretation of an ontology differs from the philosophical understanding. While ontology for a philosopher is *a particular system of categories accounting for a certain vision of the world* (GUARINO, 1998), independent of a particular language, for the artificial intelligence researcher, it refers to a *particular artifact constituted by a specific vocabulary* (NECHES *et al.*, 1991, CHANDRASEKARAN *et al.*, 1999) that describes a certain domain by explicitly constraining the intended meaning of the vocabulary words (JAZIRI & GARGOURI, 2010).

According to JAZIRI & GARGOURI (2010), the common understanding of all the definitions and interpretations of an ontology orbit around two main characteristics: formality and consensus. All of the ontology definitions accentuate the importance of representing the knowledge in a consensual manner. Not the same thing can be said about formality requirements because some authors choosing different requirements. Nevertheless, the general vision is that ontologies should be machine-readable, if not directly human readable, they

should at least contain plain text notices or explanations of concepts and relations to the human user.

While analyzing the above most relevant definitions of ontology, JAZIRI & GARGOURI (2010) assert that there is consensus among the ontology community and so there is not confusion about its usage. Different definitions provide different and complementary points of view of the same reality. The authors conclude their analysis by asserting that ontologies aim to capture consensual knowledge in a generic and formal way, and that they may be reused and shared across applications and by groups of people.

JAZIRI & GARGOURI (2010), GANDON (2010) and GUIZZARDI (2005, Chapter 3) provide a more complete compilation of definitions as well as a deeper discussion about their differences and similarities. For this dissertation I will use (STUDER *et al.*, 1998) definition as a point of reference because how it adopts the most important characteristics an ontology can have and how each characteristic is thoroughly explained.

Regardless of the domain they are used for, ontologies are essentially knowledge artifacts. This means that the properties they have and the advantages they bring as knowledge artifacts apply when being used as software artifacts in CS or as an artifact in another specific domain. Therefore, in the next section, I will first present the goals that ontologies aim to achieve as knowledge artifacts and the advantages they bring when applied in a particular domain.

2.4 – Ontologies as Knowledge Artifacts

First, the question “Why would someone want to develop an ontology?” made by NOY & MCGUINESS (2001) need to be answered. In order to do so, I will present the goals ontologies aim to achieve and I will explain how ontologies help to achieve them.

- **Knowledge sharing and reuse**

The two most important goals of ontologies that I identified in my research are *knowledge sharing* and *knowledge reuse*. Those terms appeared in the articles of GRÜNINGER & FOX (1995), CHANDRASEKARAN *et al.* (1999), CORCHO *et al.* (2003, 2006), NOY & MCGUINNESS (2001), RUIZ & HILERA (2006), GAŠEVIC *et al.* (2009b) and many others.

GAŠEVIC *et al.* (2009b) claim that the major purpose of ontologies is not to serve as vocabularies and taxonomies; but to provide *knowledge sharing* and *knowledge reuse* to applications. Therefore, ontologies provide a description of the concepts and relationships that can exist in a domain and that can be shared and reused among intelligent agents and applications. This reuse can provide the basis for *semantic interoperability* between different systems (GANDON, 2010). *Semantic interoperability* is a knowledge-level concept that provides the ability to bridge semantic conflicts arising from differences in implicit meanings, perspectives, and assumptions (JAZIRI & GARGOURI, 2010).

According to GAŠEVIC *et al.* (2009b) and USCHOLD & GRÜNINGER (1996), knowledge sharing and reuse is still not easy in practice, even if an ontology is readily available for a given purpose, which severely limits *interoperability*. First, there are several different languages for representing ontologies, and knowledge base development tools may not support the language used to develop the ontology. Second, competing approaches and working groups that create different technologies, traditions, and cultures. Third, different ontologies have been developed to describe the same topic or domain. Finally, the reusability-usability trade-off problem applied to the ontology field states that the more reusable an ontology is, the less usable, it is, and vice versa. It means that the more generic ontologies are, the more reusable they become, because they do not make any commitment to a particular domain. However, at the same time, applying such ontology in a particular application requires considerable refinement and adaptation (GÓMEZ-PÉREZ & BENJAMINS, 1999).

- **Ambiguity elimination**

According to RUIZ & HILERA (2006) and GANDON (2010), ontologies are considered a powerful tool to reduce conceptual and terminological ambiguity. They specify terms with unambiguous meanings, with semantics independent of reader and context. Translating the terms in an ontology from one language to another does not change it conceptually. Thus, an ontology provides a vocabulary and a machine processable common understanding of the topics that the terms denote. The meanings of the terms in an ontology can be communicated between users and applications (GAŠEVIC *et al.*, 2009b).

- **Clarifying the knowledge structure**

The advantage of clarifying the knowledge structure is that it makes the domain assumptions explicit. Implicit knowledge made explicit (NOY & MCGUINNESS, 2001, RUIZ & HILERA, 2006). Any knowledge base built is based on a conceptualization that is usually owned by its builders and it is usually implicit. An ontology is an explicit representation of the very implicit knowledge. It contributes greatly to knowledge reuse and sharing considering that implicit knowledge prevents sharing and reuse (MIZOGUCHI, 2001).

- **Ease of communication**

USCHOLD & JASPER (1999) claim that, fundamentally, ontologies are used to improve communication between either human or computers. Since they provide a consensual vocabulary, it solves problems like the lack of a *shared understanding* that leads to *poor communication* within and between these people and their organizations that arise because different needs and backgrounds contexts, widely varying viewpoints and assumptions regarding what is essentially the same subject matter. Each uses different jargon; each may have differing, overlapping and/or mismatched concepts, structures and methods (USCHOLD & GRUNINGER 1996). When all participants in the communication process commit to the definitions provided by the ontology vocabulary, it eliminates those problems by providing an agreed communication protocol (STUDER *et al.*, 1998).

- **Metamodel**

Being shared world models, content theories, representational artifacts of essential knowledge about topics and domains, and reusable building blocks of knowledge-based systems, ontologies are also tightly coupled to other concepts related to domain/world modeling, such as metadata and metamodeling. A metamodel is an explicit model of the constructs and rules needed to build specific models within a domain of interest (GAŠEVIC *et al.*, 2009b).

MIZOGUCHI (2001) claims that ontologies provide meta-model functionality. A model is usually built in the computer as an abstraction of the domain. And, an ontology provides us with concepts and relations among them which are used as building blocks of the

model. Thus, an ontology specifies the models to build by giving guidelines and constraints which should be satisfied.

- **Domain knowledge, operational knowledge and instance knowledge**

NOY & MCGUINESS (2001) assert that ontologies help separate domain knowledge from the operational (reasoning) knowledge. While ontologies are the repositories of the declarative knowledge and rules of the domain, *problem solving methods* (PSMs) specify the reasoning to solve concrete problems in a procedural way (JAZIRI & GARGOURI 2010, GÓMEZ-PÉREZ & BENJAMINS, 1999).

WONGTHONGTHAM *et al.* (2009) note that the domain knowledge is separate from the instance knowledge. The instance knowledge varies depending on its use for a particular project. The domain knowledge is quite definite, while the instance knowledge is particular to the problem domain and developmental domain in a project.

In the next section, I will present the potential applications of ontologies in the areas of KE, CS and Software Engineering as well as examples of existing ontologies.

2.5 – Applications of Ontologies

Ontologies have become a major conceptual backbone for a broad spectrum of applications. They are not developed just for knowledge-based systems, but for all software systems – all software needs models of the world, and hence can make use of ontologies at design time (CHANDRASEKARAN *et al.*, 1999). The major application fields for ontologies nowadays include knowledge management, e-learning, e-commerce, and integration of Web resources, intranet documents, and databases. They also include cooperation of Web services with enterprise applications, natural-language processing, bio-informatics tools, intelligent information retrieval (especially from the Internet), virtual organizations, and simulation and modeling (GAŠEVIC *et al.*, 2009b, GANDON, 2010).

Ontologies are very popular mainly within research fields that require a knowledge-intensive approach to their methodologies and system development, such as knowledge engineering (GRUBER, 1993, USCHOLD & GRUNINGER, 1996, GUARINO, 1998, GÓMEZ-PÉREZ & BENJAMINS 1999), knowledge representation, qualitative modeling,

language engineering, database design, information modeling, information integration, knowledge management and organization etc. (JAZIRI & FAIEZ, 2010).

According to USCHOLD & TATE (1998), ontologies have been used within commercial applications already. However, details are rarely disseminated. This is because commercial sensitivity, or the fact that applications are often deeply embedded, so it is difficult to demonstrate clearly the important benefits of an ontology in a practical context.

Although an ontology can possibly be used as a solution to represent all the concepts and the relationships characterizing a specific field (JAZIRI & GARGOURI, 2010), the use of one ontology for all application contexts will never be possible. Neither will an ontology be suitable for all subjects and domains nor will such a large and heterogeneous community as the Web community ever agree on a complex ontology for describing all their issues (FENSEL, 2003).

GANDON (2010) lists some existing ontologies and their subjects. Table 1 provides a list with some of these. Note how different the domains they describe are from each other.

Table 1 – List of some existing ontologies

Ontology	Description
Bibliographic ontology	Reuses data types taken from ISO standard.
SUMO[2]	It is the largest formal public ontology in existence today. Written in SUO-KIF. It is free and owned by the IEEE.
CHEMICALS ontology	Contains knowledge within the domain of chemical elements and crystalline structures.
CoreLex	Ontology for lexical semantic database and tagset nouns.
EngMath	Contains ontologies for mathematics and engineering.
Gene Ontology	An ontology for molecular functions, biological processes and cellular components.
Gentology	Ontology for genealogy applications
Open Cyc	An upper ontology containing concepts of common knowledge.

[2] http://www.adampease.org/OP/

2.5.1 – Ontologies in Knowledge Engineering

The role of ontologies in the KE process is to facilitate three activities: the construction or modelling of the domain knowledge into a domain model; the analysis of the domain knowledge and its implementation. They also influence problem-solving knowledge (STUDER *et al.*, 1998).

Knowledge management is a popular and commercially successful application of KE (STUDER *et al.*, 1998). Knowledge management is concerned with acquiring, maintaining, and accessing knowledge of an organization. It aims to exploit an organization's intellectual assets for greater productivity, new value, and increased competitiveness (FENSEL, 2003).

GAŠEVIC *et al.* (2009c) claim that the Semantic Web has been one of the hottest R&D topics in recent years in the AI community, as well as in the Internet community – the Semantic Web is an important W3C activity. According to BERNERS-LEE (2000) and BERNERS-LEE *et al.* (2001), the Semantic Web is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.

GAŠEVIC *et al.* (2009c) assert that the Semantic Web is about how to implement reliable, large-scale interoperation of Web services, to make such services computer interpretable – to create a Web of machine-understandable and interoperable services that intelligent agents can discover, execute, and compose automatically.

Ontologies play multiple roles in the architecture of the Semantic Web (GAŠEVIC *et al.*, 2009c):

- They enable Web-based knowledge processing, sharing, and reuse between applications, by the sharing of common concepts and the specialization of the concepts and vocabulary for reuse across multiple applications;
- They establish further levels of interoperability (semantic interoperability) on the Web in terms of mappings between terms within the data, which requires content analysis;
- They add a further representation and inference layer on top of the Web's current layers;

2.5.2 – Ontologies in Computer Science

In CS, an ontology is a software artifact. It is a computer representation of chosen properties of existing things; this representation being usually done in a formalism allowing for some form of rational and automated reasoning. An ontology is the result of an exhaustive and rigorous formulation of the conceptualization of a domain. This conceptualization is often said to be partial because it is impossible that one could capture the full complexity of a domain in such formalisms (GANDON, 2010).

Formerly reserved to expert systems that simulate human reasoning in specific areas, ontologies are now integrated in a large family of information systems. They are used to: describe and deal with multimedia resources; ground the interoperability of network applications; pilot automatic processing of natural language; build multilingual and intercultural solutions; allow integration of heterogeneous sources of information; describe complex interaction protocols; check the consistency of models; support temporal and spatial reasoning; make logical approximations; and so on (GANDON, 2010).

In the context of building an information system, this lack of a shared understanding leads to difficulties in identifying requirements and thus in the definition of a *specification* of the system (USCHOLD & GRUNINGER, 1996). The lack of a precise and complete *specification* is harmful for the development of an information system because its modeling requires a perfect knowledge of the studied domain and a deep analysis of the user's requirements. This task becomes very difficult because the current applications become increasingly complex and use an enormous quantity of concepts coming from heterogeneous sources (JAZIRI & GARGOURI, 2010). Another major hindrance to successful system development projects is the lack of consistent terminology. Since systems development is a collaborative activity, involving not only system developers but also domain experts and user representatives, the understanding of each other is a prerequisite for an effective collaboration (HALLBERG *et al.*, 2014). Finally, any software that does anything useful cannot be written without a commitment to a model of a relevant world, i.e., commitments to entities, properties, and relations in that world (CHANDRASEKARAN *et al.*, 1999, GUIZZARDI *et al.*, 2002).

The introduction of an ontology in an information system aims at reducing or even eliminating the conceptual and terminological confusion and at aligning our understanding in

order to improve communication, sharing, interoperability and the degree of possible reuse. Ontologies in computer science offer a unifying framework and provide primitives i.e., basic elements, building blocks for improving communication between people, between people and systems, and between systems (GANDON, 2010). Well-structured and well-developed ontologies enable various kinds of consistency checking from applications (e.g., type and value checking for ontologies that include class properties and restrictions) (GAŠEVIC *et al.*, 2009b).

2.5.3 – Ontologies in Software Engineering

Software Engineering (SE) is the "application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software". In order to cope with the complexity inherent to software, there has been a constant drive to raise the level of abstraction through modeling and higher-level programming languages (AHMED, 2008).

One of the main benefits of the use of ontologies in software development is the opportunity to adopt a reuse-based approach to the requirements engineering. In traditional SE, for each new application to be built, a new conceptualization is developed. This approach is extremely useful since elicitation is the activity that requires most effort in the software development. Experts are scarce and costly resources but they are essential to this activity. Therefore, it is important to share and reuse the captured knowledge (GUIZZARDI *et al.*, 2002).

There are many works in the SE community that identify places in software cycle (requirement elicitation, for example) where ontologies can contribute to improve the current state of SE (GAŠEVIĆ *et al.*, 2009, HAPPEL & SEEDORF, 2006, AHMED, 2008).

The system engineering benefits when using ontology-based development can be summarized as follows (USCHOLD & JASPER, 1999, USCHOLD & GRUNINGER, 1996):

- Communication: ontologies allow for the reduction of conceptual and terminological ambiguity, as they provide us with a framework for unification. Ontologies also permit an increased consistency, eliminating ambiguity and integrating distinct user viewpoints

- Interoperability: When different users or systems need to exchange data or when different software tools are used, the concept of interoperability has some important repercussions. In this sense, the ontologies can act as an "interlingua", that is, they can be used to support the translation between different languages and representations, as it is more efficient to have a translator for each part involved (with an exchange ontology) than to design a translator for each pair of involved parts (languages or representations).
- Reusability: the ontology encodes domain information (including software components) in such way that sharing and reuse are possible.
- Search: an ontology may be used as metadata serving as an index into a repository of information.
- Knowledge acquisition: the ontology guides knowledge acquisition if it is used as the starting point of the knowledge acquisition process because it provides terms and concepts of the domain the user is researching.
- Reliability: the ontology allows the automation of consistency checking resulting in more reliable software.
- Specification: the ontology can assist the process of identifying requirements and defining specification for an IT system.
- Maintenance: use of ontologies in system development, or as part of an end application, can render maintenance easier in a number of ways. Systems which are built using explicit ontologies serve to improve software documentation which reduces maintenance costs.

In the next section, I will present the components that make an ontology.

2.6 – What Are Ontologies Made Of?

Ontologies define with different levels of formality the meaning of the concepts (terms) and the relationships between them. The concepts and relationships defined in the ontology form a vocabulary that is used to model the domain (STUDER *et al.*, 1998).

They are usually organized in taxonomies which are a hierarchical organization of the relevant concepts and the relevant relationships between these concepts, as well as rules and axioms that constrain these representations (GANDON, 2010). A good taxonomy should separate its corresponding entities into mutually exclusive, unambiguous groups and subgroups that, taken together, include all possibilities. It should also be simple, easy to remember, and easy to use. Every ontology provides a taxonomy in a machine-readable and machine-processable form (GAŠEVIC *et al.*, 2009b). It is important not to confuse ontologies and taxonomies. Ontological knowledge goes far beyond the taxonomical knowledge since it is a full specification of a domain with complete formal definitions of the concepts of the domain (GANDON, 2010).

The set of all the properties of a concept is called the intension of the concept, and the set of all the objects or beings that are instances of this concept is called the extension of the concept. In ontologies, the intensions are usually organized in a taxonomy or hierarchy of types. The act of placing a class below another is called subsumption. It is also the name of the link between a sub-category and a parent category. The importance of taxonomic organization is justified by the fact that classification and identification and categorization are very common inferences that we use all day long (GANDON, 2010).

Despite the representation language being used, ontologies share a common set of characteristics and components in order to make knowledge representation and inference tasks possible. The following components of ontologies are: concepts, slots, relationships, axioms, instances and operations (JAZIRI & GARGOURI, 2010).

- Concepts: Also called classes. They are the description of the common features that a set of individuals/objects have. Concepts are general, abstract or concrete notions within a domain of discourse (GÓMEZ-PÉREZ & BENJAMINS, 1999, NOY & MCGUINNESS, 2001). They are similar to the classes in the object-oriented modeling paradigm. A concept can have sub-concepts using inheritance relationships. In the frame-based knowledge representation paradigm, metaclasses can also be defined. Metaclasses are classes whose instances are classes. They usually allow for gradations of meaning, since they establish different layers of classes in the ontology where they are defined. (CORCHO *et al.*, 2006)

- Relations: Represent a type of association between concepts of the domain. Ontologies usually contain binary relations. The first argument is known as the domain of the relation, and the second argument is the range. Binary relations are sometimes used to express concept attributes (slots). Attributes are usually distinguished from relations because their range is a datatype (CORCHO *et al.*, 2006).
- Slots: Also called properties, attributes or roles. They describe the various features and attributes of a concept (and its instances). Facets (sometimes called role restrictions) describe restrictions on slots (NOY & MCGUINNESS, 2001).
- Axioms: Also called constraints. Are formal sentences that are always true (GUARINO, 1998, GÓMEZ-PÉREZ & BENJAMINS, 1999). They specify some constraints on the ontology elements (they constrain their interpretation) and are normally used to represent knowledge that cannot be formally defined by the other components. In addition, they are used to verify the consistency of the ontology itself or the consistency of the knowledge stored in a knowledge base and to infer new knowledge (CORCHO *et al.*, 2006).
- Instances: They are individuals holding concept definitions and facts representing relationships between individuals. An ontology together with a set of individuals instances of classes constitutes a knowledge base (NOY & MCGUINNESS, 2001, GÓMEZ-PÉREZ & BENJAMINS, 1999).
- Operations/Functions/Rules: Rules are generally used to infer knowledge in the ontology, such as attribute values, relation instances, etc. Ontological representation languages enable the execution of a certain basic set of operations to cover updating and querying tasks on ontologies. Likewise, new concepts can be defined, properties related to concepts and values changed or added during the entire life of the ontology.

In the next section, I will present the difference between ontologies, models and data models.

2.7 – Differences between Ontologies and Models

In the SE and Information Systems (IS) communities, perhaps because of the historical importance of conceptual modeling, exists frequent confusion between ontology and conceptual models (AHMED, 2008). It is important to establish the difference between them.

A most fundamental feature of a model is that it can be descriptive or prescriptive (SEIDEWITZ, 2003). In the former case, the model describes reality, but reality is not constructed from it. In the latter case, the model prescribes the structure or behavior of reality and reality is constructed according to the model; that is, the model is a specification of reality (AßMANN *et al.*, 2006).

AßMANN *et al.* (2006) defines that an ontology is a shared, descriptive, structural model, representing reality by a set of concepts, their interrelations, and constraints under the open-world assumption. They also define that a specification model is a prescriptive model, representing a set of artifacts by a set of concepts, their interrelations, and constraints under the closed-world assumption. Finally, they assume that specification models focus on the specification, control, and generation of systems; while ontologies focus on description and conceptualization (structural modelling) of things. Both kinds of models have in common the qualities of abstraction and causal connection.

SPYNS *et al.* (2002) establish that the main difference between the data models and ontologies is that while the former are task specific and implementation oriented, the latter should be as much generic and task independent as possible. Also, they assert that, unlike data models, the fundamental asset of ontologies is their relative independence of particular applications, i.e., an ontology consists of relatively generic knowledge that can be reused by different kinds of applications/tasks.

There are several important differences between ontologies and database schemas described by FENSEL (2003) and RUIZ & HILERA (2006):

- Languages for defining and representing ontologies (OWL, KIF, etc.) are syntactically and semantically richer than common approaches for databases (SQL, etc.).
- The knowledge that is described by an ontology consists of semi-structured information (that is, text in natural language) as opposed to the very structured data of the database (tables, classes of objects, etc.).

- An ontology must be a shared and consensual conceptualization because it is used for information sharing and exchange. Identifiers in a database schema are used specifically for a concrete system and do not have the need to make an effort to reach the equivalent of ontological agreements. Also, the conceptualization and the vocabulary of a data model are not intended a prori to be shared by other applications (SPYNS *et al.*, 2002).
- An ontology provides a domain theory and not the structure of a data container.

An important property of ontologies is the so-called open-world assumption (OWA). It states, intuitively, that anything not explicitly expressed by an ontology is unknown. Hence, ontologies use a form of partial description or under-specification as an important means of abstraction. In contrast, most system models underlie the closed-world assumption (CWA) which what has not been specified as true must be false, to restrict arbitrary extensions of the system, which could introduce inconsistencies (AßMANN *et al.*, 2006, SEQUEDA, 2012).

According to SEQUEDA (2012), OWA applies when a system has incomplete information. This is the case when we want to represent knowledge (ontologies) and want to discover (infer) new information. For example, consider a patient's clinical history system. If the patient's clinical history does not include a particular allergy, it would be incorrect to state that the patient does not suffer from that allergy. It is unknown if the patient suffers from that allergy, unless more information is given to disprove the assumption.

On the other hand, the CWA applies when a system has complete information. This is the case for many database applications. For example, consider a database application for airline reservations. If a passenger is looking for a direct flight between Austin and Madrid, and it doesn't exist in the database, then the result is "There is no direct flight between Austin and Madrid." For this type of application, this is the expected and correct answer (SEQUEDA, 2012).

Diego Calvanese[3] states that the CWA does not refer to the fact that your inference problem can have only two answers, but to the fact that the objects in the domain of discourse

---

[3] https://www.linkedin.com/grp/post/119766-6026870883128270850. You must be registered in the group to see the post.

are only those that are explicitly mentioned in your knowledge base. In other words, under the CWA, you cannot infer the existence of new objects. Even so, CWA can be used for reasoning.

Diego provides an example that illustrates the difference of how reasoning works in CWA and OWA. Suppose you assert an axiom stating that every person has a father, who is a person, and that the only person you know of is John. If someone asks for the name of those persons who have a grand-grand-father, by using the axiom you can infer that "John" is a correct answer, even if you don't know who the grand-grandfather of John is. Now, if you also ask for the name of the grand-grandfather of John, under the OWA you cannot return any answer, since this is an unknown object in your domain of discourse. Instead, under the CWA, since the only person you know of is John, and the grand-grandfather of John must exist and be a person (due to the axiom), you can infer that the grand-grandfather of John must be John itself.

Sequeda (2012) also provides a simple example that illustrates another difference between OWA and CWA. Consider the following statement: "Juan is a citizen of the USA". The answer to "Is Juan a citizen of Colombia?" under a CWA is no, whereas under the OWA is unknown. Now, the following statements are true: "a person can only be citizen of one country" and "Juan is a citizen of Colombia". In a CWA system, this would be an error because we previously stated that person can only be a citizen of one country and we assume that USA and Colombia are different countries. In an OWA system, instead of generating an error, it would infer the following logic: "If a person can only be citizen of one country, and if Juan is a citizen of USA and Colombia, then USA and Colombia must be the same thing".

In the CWA case, it is assumed that USA and Colombia are different countries. With OWA, this is not assumed. This is called the Unique Names Assumption (UNA). CWA systems have UNA but OWA systems do not. However, one could manually add the UNA. In other words, if I have a list of all the countries, I would have to explicitly state that each country is different from each other. In our example, if we add the following statement: "USA is different from Colombia," the OWA would now generate an inconsistency (SEQUEDA, 2012).

In the next section, I will present the different types of ontologies.

2.8 – Types of Ontologies

GUARINO (1997) establishes that an ontology can be classified depending on the level of detail. It allows distinguishing 2 types of ontology: reference ontologies (off-line) and shareable ontologies (on-line). While a fine-grained ontology will specify more precisely the intended meaning of a vocabulary (and therefore it can be used off-line for reference purposes), it would be difficult to be assembled and reasoned on it (GUARINO, 1998). On the other hand, a coarse (shareable) ontology would be much more easily shared among its clients that *already agree on the underlying conceptualization* (GUARINO, 1998), and therefore it can be used on-line to support the system's services.

GUARINO (1997) also claims that *the level of dependence* on a particular task or point of view allows us to distinguish:

- Top-level ontologies: specify very general concepts, *which are independent of a particular problem or domain* (GUARINO, 1998);
- Domain ontologies: specializes the general concepts (of top-level ontologies), referring to a generic domain;
- Task ontologies: domain ontologies and task ontologies specialize the concepts (of top-level ontologies), referring to a generic task or activity (GUARINO, 1998);
- Application ontologies: level further specialization is involved by describing concepts *depending on a particular domain or task* and is often *roles* of domain or task entities performed during a certain activity (GUARINO, 1998).

GÓMEZ-PÉREZ & BENJAMINS (1999) simplifies the classification proposed by GUARINO (1997) by asserting that meta-ontologies (top-level), domain ontologies and application ontologies capture static knowledge in a problem-solving independent way, whereas PSMs ontologies, task ontologies and domain-task ontologies are concerned with problem solving knowledge.

While some authors request for an ontology language to be a formal one, USCHOLD & GRÜNINGER (1996) adopt a *weak* position regarding the formality requirement. The classifications are as follows:

- Highly informal: Ontologies expressed in loosely natural language;

- Structured informal: Ontologies expressed in a restricted and structured form of natural language, greatly increasing clarity by reducing ambiguity (USCHOLD & GRUNINGER, 1996);
- Semi-formal: Ontologies expressed through an artificial language;
- Rigorously formal: Ontologies are expressed by meticulously defined terms with formal semantics, theorems and proofs of such properties as soundness and completeness (USCHOLD & GRUNINGER, 1996).

There is also the distinction to be made between lightweight and heavyweight ontologies. It is a simplification of the classification based on the level of richness of their internal structure, whereby lightweight ontologies will be principally taxonomies, while heavyweight ontologies are those which model a certain knowledge "in a deeper way and provide more restrictions on domain semantics". The former include concepts, concepts taxonomies, relationships between concepts, and properties that describe these concepts. The latter add axioms and constraints, in order to clarify the meaning of terms (RUIZ & HILERA, 2006). There are much more types of classifications of ontologies. Check JAZIRI & GARGOURI (2010) and RUIZ & HILERA (2006) for a deeper discussion on the matter.

Now that I presented what ontologies are and what are they made of, I can now talk about how one does build an ontology. In the next section, I will present the field of ontological engineering.

2.9 – Ontological Engineering

Ontological engineering is a successor of knowledge engineering which has been considered a technology for building knowledge-intensive systems. Although knowledge engineering has contributed to eliciting expertise, organizing it into a computational structure, and building knowledge bases, AI researchers have noticed the necessity of a more robust and theoretically sound engineering which enables knowledge sharing/reuse and formulation of the problem solving process itself (MIZOGUCHI, 2001). Also, to develop a really useful ontology requires a lot of engineering effort, discipline, and rigor (GAŠEVIC *et al.*, 2009b).

Ontological engineering denotes a set of design principles, development processes and activities, supporting technologies, and systematic methodologies that facilitate ontology

development and use throughout its life cycle – design, implementation, evaluation, validation, maintenance, deployment, mapping, integration, sharing and reuse (GAŠEVIC *et al.*, 2009b).

Unlike software engineering where there are well-documented standards for the domain (SWEBOK - *Software Engineering Body of Knowledge*), there is no definition and standardization of the life cycle of an ontology, methodologies and techniques that drive the development of ontologies. GÓMEZ-PÉREZ *et al.* (1996) assert that ontological engineering is a craft rather than a science.

In the next section, I will present the design principles that an ontology engineer must follow to ensure that the developed ontology satisfies certain criteria. According to (STUDER *et al.*, 1998), an ontology is optimal if it satisfies as much as possible all design principles.

2.10 – Ontology Design Principles

Ontology design principles are objective criteria for guiding and evaluating ontology designs. GRUBER (1995) identified the following five principles:

- Clarity and Objectivity: An ontology should communicate effectively the intended meaning of defined terms. Definitions should be objective. Definitions can be stated on formal axioms, and a complete definition (defined by necessary and sufficient conditions) is preferred over a partial definition (defined by only necessary or sufficient conditions). All definitions should be documented with natural language.
- Minimal encoding bias: The conceptualization should be specified at the knowledge level without depending on a particular symbol-level encoding. Encoding bias should be minimized, because knowledge-sharing agents may be implemented in different representation systems and styles of representation.
- Maximum Monotonic Extendibility: An ontology should be designed to anticipate the uses of the shared vocabulary. One should be able to define new terms for special uses based on the existing vocabulary, in a way that does not require the revision of existing definitions.

- Coherence: An ontology should be coherent: that is, it should sanction inferences that are consistent with the definitions. If a sentence that can be inferred from the axioms contradicts a definition or example given informally, then the ontology is inconsistent.
- Minimal ontological commitments: An ontology should make as few claims as possible about the world being modeled, allowing the parties committed to the ontology freedom to specialize and instantiate the ontology as needed. According to this principle, we should not commit to a specific format for dates, for currencies, etc., when designing our ontologies, since such details could be different in different systems (CORCHO *et al.*, 2006). This principle assures maximum reusability, but there is a well-known trade-off between reusability and usability (the more reusable, the less usable, and vice versa) (STUDER *et al.*, 1998).

SMITH (2006) lists fourteen principles for ontology design. Some of them are:

- Openness: An ontology should be open and available to be used by all potential users without any constraint;
- Reusing available resources: An ontology should utilize recognized resource that already deals with entities and operators that the ontology covers;
- Intelligible definitions: Use definitions which are both humanly intelligible (to avoid error in human use) and formally specifiable (in order to support a type of software);
- Non-subjective definitions: When formulating definitions avoid the use of phrases that invite subjective interpretations, for example “X **may** be something…”

Some other principles have proven useful in ontology design, such as:

- Completeness (GÓMEZ-PÉREZ & BENJAMINS, 1999), which means that a definition expressed in terms of necessary and sufficient conditions is preferred over a partial definition (defined only through necessary or sufficient condition).
- Modularity to minimize coupling between modules (GÓMEZ-PÉREZ & BENJAMINS, 1999). It allows more flexibility and a variety of uses, specialization of general concept into more specific concepts, classification of concepts the similar features to guarantee according to inheritance of such features, and standardized name

conventions (FENSEL, 2003). As separated modules, it is also possible to “compile ontologies” and optimize the inferences they support (GANDON, 2010).

- The standardization of names, which proposes to use the same conventions to name related terms, in order to ease the understanding of the ontology (CORCHO *et al.*, 2006).

Ontology design, like most design problems, will require making tradeoffs among the criteria. An apparent contradiction is between extendibility and ontological commitment. An ontology that anticipates a range of tasks need not include vocabulary sufficient to express all the knowledge relevant to those tasks (requiring an increased commitment to that larger vocabulary). An extensible ontology may specify a very general theory, but include the representational machinery to define the required specializations (GRUBER, 1995).

NOY & MCGUINNESS (2001) conclude their work by asserting that ontology design is a creative process and no two ontologies designed by different people would be the same. The potential applications of the ontology and the designer’s understanding and view of the domain will undoubtedly affect ontology design choices. The quality of the ontology can only be assessed by using it in applications.

Before using an ontology, we have to build it, beginning from existing applications or from scratch. To build ontologies, several basic questions arise related to the methodologies, languages and tools to be used in its development process (CORCHO *et al.*, 2003):

- Which methods and methodologies can be used for building ontologies?
- Which tools support the ontology life-cycle stages?
- Which language should be used to formalize and implement an ontology?

Therefore, in the next section, I will talk about the ontology development process by detailing the similarities and differences between the several methodologies that exist.

2.11 – Ontology Development Methodologies

An ontology development methodology comprises a set of established principles, processes, practices, methods, and activities used to design, construct, evaluate, and deploy ontologies. Several such methodologies have been reported in the literature (USCHOLD &

KING, 1995, GRÜNINGER & FOX, 1995, FERNÁNDEZ-LÓPEZ *et al.*, 1997, NOY & MCGUINNESS, 2001, GAŠEVIC *et al.*, 2009b). As a consequence, the construction of an ontology cannot be conducted in an improvised manner (JAZIRI & GARGOURI, 2010).

There are several surveys about ontology development methodologies (FERNÁNDEZ-LÓPEZ AND GÓMEZ-PÉREZ, 2002, CORCHO *et al.*, 2003, SURE *et al.*, 2009, GAŠEVIC *et al.*, 2009b). The following conclusions can be made about ontology development methodologies:

- None of the approaches covers all the processes involved in ontology building. Most ontology development methodologies that have been proposed to build ontologies are focused on the development activities, especially on ontology conceptualization and ontology implementation, and they do not pay too much attention to other important aspects related to management, learning, merging, integration, evolution and evaluation of ontologies;
- Some methodologies build on general software development processes and practices and apply them to ontology development;
- There are also methodologies that exploit the idea of reusing existing ontological knowledge in building new ontologies;
- Some of the more recently proposed methodologies are based on the idea of using publicly available community-based knowledge to simplify and speed-up the development of ontologies;
- Most of the approaches present some drawbacks in their uses. Some of them have not been used by external groups and, in some cases; they have been used in a single domain;
- Most of the approaches do not have a specific tool that gives them technological support. Besides, none of the available tools covers all the activities necessary in ontology building;
- There is no consensus about the best practices to adopt concerning the construction of an ontology.

According to NOY & MCGUINNESS (2001), ontology development is different from designing classes and relations in object-oriented programming. Object-oriented programming centers primarily around methods on classes – a programmer makes design decisions based on the operational properties of a class, whereas an ontology designer makes these decisions based on the structural properties of a class. As a result, a class structure and relations among classes in an ontology are different from the structure for a similar domain in an object-oriented program.

NOY & MCGUINNESS (2001) state that there is not a one correct methodology for developing ontologies. However there are some fundamentals rules in ontology design that can help the developer to make wise design decisions. These are given as follows:

- There is no one correct way to model a domain, there are always viable alternatives. The best solution almost always depends on the application that one has in mind and the extensions that are anticipated;
- Ontology development is necessarily an iterative process;
- Concepts in the ontology should be close to objects (physical or logical) and relationships in the domain of interest. These are most likely to be nouns (objects) or verbs (relationships) in sentences that describe the domain.

GAŠEVIC *et al.* (2009b) make two important observations from their brief survey of ontology development methodologies. First, there are many common points in the various methodologies. Step in different processes may be named differently, may also be of different granularity, or may only partially overlap; but the processes are still very much alike. Second, many of the principles and practices of ontology development are analogous to those of software engineering.

The ontology development process does not identify the order in which the activities should be performed (FERNÁNDEZ-LÓPEZ *et al.*, 1997). This is the role of ontology life cycle, which identifies the set of stages through which the ontology moves during its lifetime, describes which activities are performed during each stage and how the stages are related (FERNÁNDEZ-LÓPEZ *et al.*, 1997, CORCHO *et al.*, 2006). Each ontology building methodology may have a different cycle with different stages. Those activities and the life

cycle will be detailed more thoroughly in Chapter 5, where the building methodology used will be detailed.

2.12 – Ontology Development Tools

The standard tool set of an ontology engineer includes ontology representation languages and graphical ontology development environments. More recently, ontology learning tools have also started to appear, in order to partially automate the development process and help in evolution, updating, and maintenance of ontologies. Other tools are also required in the context of developing ontologies for deployment on the Semantic Web (GAŠEVIC *et al.*, 2009b).

Ontology tools appeared, in the mid-1990s, and can be classified in the following two groups (CORCHO *et al.*, 2006):

- Tools whose knowledge model maps directly to an ontology language, hence developed as ontology editors for that specific language.
- Integrated tool suites whose main characteristic is that they have an extensible architecture, and whose knowledge model is usually independent of ontology languages. These tools provide a core set of ontology-related services and are easily extended with other modules to provide more functions.

2.12.1 – Ontology Representation Languages

Ontology representation languages are knowledge representation languages. Therefore, they should be capable of both syntactic and semantic representation of entities, events, actions, processes, and time. However, not all of the existing knowledge representation languages have support for all of these things. Also, each existing language supports some, but not all, popular knowledge representation techniques. In addition, some knowledge representation languages are designed to provide support for knowledge communication and interchange between intelligent systems (GAŠEVIC *et al.*, 2009a).

There are a number of ontology representation languages around. Some of them were developed at the beginning of the 1990s within the AI community. Others appeared in the late 1990s and later, resulting from the efforts of AI specialists and the World Wide Web

Consortium (W3C). Some of the best-known examples of the early ontology representation languages are (GAŠEVIC *et al.*, 2009b):

- KIF, which is based on first-order logic;
- Ontoligua, which is built on top of KIF but includes frame-based representation;
- Loom, based on description logics.

Also, most of the recent languages were developed to support ontology representation on the Semantic Web, and hence they are also called "Semantic Web languages". Among the widely used Web-based ontology languages, the most important are (GAŠEVIC *et al.*, 2009b):

- SHOE, built as an extension of HTML;
- XOL, developed by the AI center of SRI International as an XML-ization of a small subset of primitives from the OKBC protocol called OKBC-Lite;
- RDF, developed by the W3C as a semantic-network-based language to describe Web resources;
- RDF Schema, also developed by the W3C, is an extension of RDF with frame-based primitives; the combination of both RDF and RDF Schema is known as RDF(S);
- OIL, which is based on description logics and includes frame-based representation primitives;
- DAML+OIL is the latest release of the earlier DAML (DARPA Agent Markup Language), created as the result of a joint effort of DAML and OIL developers to combine the expressiveness of the two languages;
- OWL, or Web Ontology Language, developed under the auspices of the W3C and evolved from DAML+OIL and RDF; OWL is currently the most popular ontology representation language.

For more comprehensive information and comparative studies of all of them, refer to CORCHO *et al.* (2003) and GANDON (2010).

CORCHO *et al.* (2003) assert in their studies on ontology representation languages that in the case of needing to implement an ontology, we should decide first what our application needs in terms of expressiveness and inference services, because not all of the existing languages allow representing the same components and reason in the same way. The representation and reasoning with basic information, such as concepts, taxonomies and binary relations, is not usually enough if we want to create a heavyweight ontology and make complex reasoning with it, and existing translations between languages are not good enough yet to ensure that information is not lost in the process. Hence, making a good decision of using a specific language for representing ontologies is crucial for developing an ontology-based application.

KNUBLAUCH *et al.* (2006) compare and points the similarities and differences between Semantic Web languages and object-oriented languages. According to him, the key benefits of Semantic Web languages compared to object-oriented languages are:

- Reuse and interoperability: their models can be shared among applications and on the web;
- Flexibility: their models can operate in an open environment in which classes can be defined dynamically;
- Consistency and Quality Checking across models;
- Reasoning: they possess rich expressivity supported by automated reasoning tools.

2.12.2 – Ontology Development Environments

No matter what ontology representation language is used, there is usually a graphical ontology editor to help the developer organize the overall conceptual structure of the ontology; add concepts, properties, relations, and constraints; and, possibly, reconcile syntactic, logical, and semantic inconsistencies among the elements of the ontology. In addition to ontology editors, there are also other tools that help to manage different versions of ontologies, convert them into other formats and languages, map and link ontologies from heterogeneous sources, compare them, reconcile and validate them, and merge them. Yet other tools can help acquire, organize, and visualize the domain knowledge before and during the building of a formal ontology (GAŠEVIC *et al.*, 2009b).

Graphical ontology development environments integrate an ontology editor with other tools and usually support multiple ontology representation languages. They are aimed at providing support for the entire ontology development process and for the subsequent use of the ontology (CORCHO *et al.*, 2003). There are dozens of these environments, some examples are:

- KAON2[4]: is an open source infrastructure for managing OWL-DL, SWRL, and F-Logic ontologies;
- OntoStudio[5]: commercial modeling environment for creating and maintaining ontologies;
- Ontolingua[6]: web service offered by Stanford University;
- Protégé[7]: a Java-based open source ontology browser and editor from Stanford University. Supports RDF and OWL;
- Swoop[8]: a Java-based open source OWL ontology browser and editor from the University of Maryland;
- Synaptica[9]: ontology, taxonomy and thesaurus management software. Commercial;

---

[4] http://kaon2.semanticweb.org/

[5] http://www.semafora-systems.com/en/products/ontostudio/

[6] http://www.ksl.stanford.edu/software/ontolingua/

[7] http://protege.stanford.edu/

[8] https://github.com/ronwalf/swoop

[9] http://www.synaptica.com/products/

Chapter 3 – Games

In this chapter, I will talk about video games and the relevant concepts surrounding it. First, I will analyze common characteristics of games in order to have a clear grasp of what game is and analyze common elements that are present in games to have a clear idea of what a game is made of. Second, the same analysis will be done for video games because they are a subset of games and they have their own characteristics. Third, the field of game design will be explored first since the development of a game starts with its design. Finally, I will present the details of the video game development process. The distinct phases of the process, the different profiles of professionals that compose the development team and the different types of documentation generated will be presented in order to understand how video games are made. As an addendum, games and video games are separate terms. Games refer to any kind of game and video games refer to electronic (digital) games.

3.1 – What Are Games?

According to THORN (2013) a formal study of video games should begin, perhaps, with a clear definition of the term "game" or "video game". However, he gives two reasons why these terms should be left undefined intentionally:

- There is no clear consensus about what a game is or about all things that a game must have in order to be a game. None of the existing definitions have been universally accepted for the purposes of defining the limits of game design (BRATHWAITE & SCHREIBER, 2008).
- A definition of the term "game" is largely an academic and philosophical matter that has little bearing on game development.

There are a lot of factors that influences the existence of the first reason. According to SCHELL (2014) because the idea of what a game (or any term) means will vary a bit from person to person, but mostly, we all know what a game is. ELIAS *et al.* (2012) claim there are no precise definitions of complex concepts like "game", no definitions that will include all things that people accept as games and exclude all things that people reject.

THORN (2013) is right about the second reason. Having a precise definition of what a game is will not help us develop games, since each game can have extremely different

components and dynamics between them. However, I believe that a formal study of games will benefit from an analysis of existing definitions. This analysis will lead us to characteristics (or features) that all games have in common. SCHELL (2014) performed an analysis of existing definitions and picked ten qualities out of these various definitions.

- *Games are entered willfully*. It is not a game if the person is forced to play the game. It is simply not fun participate in an activity against your will.
- *Games have goals*. Most of the games have win and lose goals. Other games allow the players to establish their own goals within the game world.
- *Games have conflict*. To achieve their goals the player must enter in conflict with other players or the own game rules. In some games the player will try to beat himself, one example of this is beating his own high score.
- *Games have rules*. This one is self-explanatory. Rules will define what players can and cannot do within the game world in order to achieve their goals. In short, the rules define the structure of a game. Without structured and well-defined rules, player will not be able to achieve the intended goals.
- *Games can be won and lost*. Not all games have win and lose conditions. However, these types of conditions greatly increase the player engagement on the game.
- *Games are interactive*. In a game, the player has an active role in changing the state of the game. The player interacts with the game and the game interacts with the player. The former is by choosing actions permitted by the rules of the game; the latter is by showing the player how the chosen actions affect the state of the game world.
- *Games have challenge*. Conflict provides some kind of challenge to the player, otherwise games can get boring pretty quick without it. Bad games have little challenge or too much challenge. Good games have just the right amount. It is worth noting that the right amount of challenge varies with the player.
- *Games can create their own internal value*. From COSTIKYAN's (2002) definition, Schell picks the term "endogenous meaning". Endogenous is a term that comes from biology, it means "caused by factors inside the organism". COSTIKYAN (2002) says that "endogenous meaning" means that things that have value inside the game have

value only inside the game. For example, Monopoly money only has meaning in the context of the game of Monopoly. The money is very important when we play the game, but, outside the game, it is completely unimportant. The more compelling a game is for a player, the greater its "endogenous value".

- *Games engage players*. Games makes players feel "mentally immersed". Games that achieve this are considered good games.
- *Games are closed, formal systems*. According to FULLERTON (2014), a system is defined as a set of interacting elements that form an integrated whole with a common goal or purpose. Schell adds that "formal" is just a way of saying that the system is clearly defined, that is, it has rules and that "closed" means that there are boundaries to the system.

The first five characteristic comes from AVEDON & SUTTON-SMITH (1971) definition "Game are an exercise of voluntary control systems, in which there is contest between powers, confined by rules in order to produce a disequilibrial outcome". The sixth, seventh and eighth characteristics comes from COSTIKYAN (2002) definition "A game is an interactive structure of endogenous meaning that requires players to struggle toward a goal".

The penultimate and last characteristics come from FULLERTON (2014) definition "A closed, formal system that engages players in structured conflict and resolves its uncertainty in an unequal outcome". She claims that at the heart of every game is a set of formal elements that, as we have seen, when set in motion, creates a dynamic experience in which players engage. According to her, the basic elements of systems are: objects, properties, behaviors and relationships. One could say that those are the most basic elements of a game.

SCHELL (2014) concludes his analysis by proposing his own definition which is "A game is a problem-solving activity, approached with a playful attitude". This definition comes from the fact that a goal of a game is clearly a problem that a player has to solve. The player has to determine the actions he can take and the obstacles he must surpass in order to achieve his goal. In short, it is a problem-solving activity. He justifies his definition in much more detail in his book *The Art of Game Design: A Book of Lenses* using the ten characteristics he found in his analysis.

An interesting game quality that SCHELL did not pick is unpredictability. ADAMS & DORMANS (2012) claim that a game that is predictable is usually not much fun as its outcome should not be clear from the start. They say that there are three ways to make a game unpredictable: include elements of chance, choices made by players and complex gameplay created by the game's rules. The most interesting is when the rules of a game are complex. They claim that complex systems usually have many interacting parts. The behavior of individual parts might me easy to understand; their rules might be simple. However, the behavior of all the parts combined can be quite surprising and difficult to foresee, the game of chess is a classic example of this effect. They conclude that complex rule systems that offer many player choices are difficult to design well.

GREGORY (2014), FULLERTON (2014), ELIAS *et al.* (2012) and TEKINBAS & ZIMMERMAN (2003) also talk about definitions of games. With the characteristics that most game can have presented, in the next section, I will present the formal elements that form the structure of a game as they are meant to be precisely defined and not offer ambiguity.

3.1.1 – What Are Games Made Of?

AVEDON & SUTTON-SMITH (1971) pose the following question: are there certain structural elements that are common to all games, regardless of the differences in games or the purpose for which the games are used, or the culture in which they are used? According to SCHELL (2014), the elements that form a game can be divided in four essential categories:

- Mechanics: These are the procedures and rules of the game. Mechanics describe the goal of the game, how players can and cannot try to achieve it, and what happens when they try. More linear entertainment experiences (movies, for example) do not possess mechanics, for it is mechanics that makes a game. The designer must choose a technology that supports the mechanics, aesthetics that emphasize them clearly to players and a story that allows your game mechanics to make sense to players. Game mechanics can be divided in seven categories: space, time, objects (its attributes and states), actions, rules, skill and chance.
- Story: It is the sequence of events that unfolds in a game. It may be linear and prescripted, or it may be branching or emergent. The designer must choose mechanics that both strengthen that story and let it emerge.

- Aesthetics: This how the game looks, sounds, smells, tastes, and feels. The designer needs to choose a technology that will allow the player to experience the aesthetics as well to amplify and reinforce them; mechanics that make the player feel like they are in the world that the aesthetics defined; and a story that let the aesthetics emerge at the right pace and have the most impact.
- Technology: It refers to any materials and interactions that make the game possible. The technology, that the designer chooses for the game, enables it do certain things and prohibit it from doing other things. The technology is essentially the medium in which the aesthetics take place, in which the mechanics occur, and through which the story will be told.

From the analysis on definitions that SCHELL performed, some elements can be deduced: rules, goals, objects, properties of objects, relationships of objects and the allowed behavior in the game world. It can be inferred that *actions* are an element of the game structure since an object has behavior. To reach the desired goal, a player must be allowed to perform actions that change the game state. Thus, *actions* are instrumental for the game structure since they are a means for the player to reach their desired goal.

Rules are the most fundamental elements of the game because they define the space, the timing, the objects, the actions, the consequence of the actions, the constraints on the actions, and the goals (SCHELL, 2014). It can be said that a game is made by its rules since everything is defined by rules. There are many types of rules. For example, goals can be considered a special kind of rule, since goals are achieved in order to reach the win state of the game. Other types are rules that restrict actions (you can only move a certain number of spaces with a piece of chess), rules that determine effects (if you capture the opponent's king you win the game of chess), boundaries (the size limits of soccer field) and outcomes (a certain action can be benefic for a player but harmful for another) (FULLERTON, 2014).

According to SCHELL (2014), every game takes place in a *space*, because of that a space can be considered an essential element of the game. It is in this space that the gameplay happens. It defines the various places that can exist in a game and how those places are related to one another. As a game mechanic, space is a mathematical construct.

Another important element of games is the interface, since it fits in the four categories proposed by SCHELL. It is the infinitely thin membrane that separates player and game. The goal of an interface is to make players feel in control of their experience. “Interface” can mean many things – a game controller, a display device, a system of manipulating a virtual character, the way the game communicates information to the player, the equipment of a board game and many other things (SCHELL, 2014). It is through the interface that the player will perform the actions that change the game state and it is through it that the player will know the overall state of the game (information may be complete or not) in order to be able to make the appropriate decisions.

A term that is always mentioned is “gameplay”. GREGORY (2014) says that “gameplay” is the action that takes place in the game, the rules that govern the virtual world in which the game takes place, the abilities of the player character(s) and the other characters and objects in the world, and the goals and objectives of the player(s). In summary, “gameplay” is the experience of interacting with the game and each experience differs depending of the game elements at action in the moment of the experience.

Concluding this section, I claim that games are composed of the following essential elements: players, objects and their relationships, actions of objects, states of objects, rules, goals that are subtypes of rules (not all rules subtype are essential), a space in which gameplay happens and an interface to receive the player input and to output the game state to the player. Finally, gameplay is the time which the player spends interacting with a combination of these elements that the game provides, this time spent by the player is an unique experience. Therefore, time is also an essential element.

Video games are a type of games. Therefore, they have their own characteristics and peculiarities. In the next section, I will talk about what video games are.

3.2 – What Are Video Games?

Before starting this section, it should be noted that the underlying properties of games and the core challenges of game design hold true regardless of the medium in which a game manifests (TEKINBAS & ZIMMERMAN, 2003).

ROGERS (2013) simply says that a video game is simply a game that is available on a video screen. There is no other definition simpler than that. However, video games are, in

essence, computer programs. Therefore, video games need further analysis on their characteristics that set them apart from other types of games, as well how it is structured as it is a software artifact.

TEKINBAS & ZIMMERMAN (2003) identify four traits that summarize the special qualities of digital games. These traits are also present in non-digital games, but digital games generally embody them more robustly. Those traits are not mutually exclusive as there is some overlap between them and they do not constitute a definitive list of traits that appear in digital games. Game designers should take advantage of those traits when creating games in a digital medium.

- **Immediate but narrow interactivity**: digital technology offer real-time game play that shifts and reacts dynamically to player decisions, thus being immediate. Interaction with a console or computer is generally restricted to an input device such as a joystick or keyboard, thus it offer narrow interactivity.
- **Manipulation of information**: digital games can store great quantities of information (rules, images, text, etc.) and manipulate them. One advantage that comes from this trait is the fact that digital games can enforce rules automatically while in board games players would have to first learn the rules of the game before playing it. Another advantage is hiding information from the player, the player will gradually learn about the game mechanics as he keeps playing the game.
- **Automated complex systems**: digital games can automate complicated procedures facilitating the play of games that would be too complicated in a non-computerized context. Examples would be physics systems and RPG battle systems; the latter is possible to play with pen and paper but would take too much time because of the amount of calculations needed.
- **Networked communication**: many digital games possess this trait that facilitates communication between players. Online multiplayer digital games are a clear example of this, they can communicate with each other through means provided by such games. Those means could be voice chat, text chat, avatar gestures, etc.

According to Jesper Juul, in a lecture titled “Play Time, Event Time, Themability”, a game is actually what computer science describes as a state machine. It contains input and

output functions, as well as definitions of what state and what input will lead to the following state (TEKINBAS & ZIMMERMAN, 2003). By looking at games as state machines, researchers can determine which rules cause the game to progress from one state from another. However, games can exist in a vast number of states making it impossible to document them all. Finite state machines are sometimes used in practice to define the behavior of simple artificially intelligent non-player characters (ADAMS & DORMANS, 2012).

Also, according to GREGORY (2014) most two- and three-dimensional video games are examples of what computer scientists would call *soft real-time interactive agent-based computer simulations*. He breaks this phrase down in order to understand what it means.

In most video games, some subset of the real world, or an imaginary world, is modeled mathematically so that it can be manipulated by a computer. The model is an approximation to and a simplification of reality (just like an ontology), because it is clearly impractical to include every detail. Hence, the mathematical model is a *simulation* of the real or imagined game world (GREGORY, 2014).

An agent-based simulation is one in which a number of distinct entities known as "agents" interact (GREGORY, 2014). Agents can be characters, vehicles, bullets and so on; the point is that those objects interact with each other causing their states to change through their actions.

All interactive video games are *temporal simulations*, meaning that the virtual game world model is *dynamic* – the state of the game changes over time as the game's events and story unfold. A video game must also respond to unpredictable inputs from its human player(s) – thus *interactive temporal simulations*. Finally, most video games present their stories and respond to player input in real time, making them *interactive real-time simulations*. One notable exception is in the category of turn-based games like non-real-time strategy games. But even these types of games usually provide the user with some form of real-time graphical user interface (GREGORY, 2014).

A "soft" real-time system is one in which missed deadlines are not catastrophic. Hence, all video games are *soft real-time systems* – if the frame rate dies, the human player generally does not. While in a *hard real-time system*, a missed deadline could mean severe

injury to or even death of a human operator. The control-rod system in a nuclear power plant is an example of one (GREGORY, 2014).

All video games are, in essence, software artifacts as they are obviously made of common components. In the next section, I will present the elements that are essential in the making of a video game.

3.2.1 – What Are Video Games Made Of?

According to THORN (2013), a video game is made from three main pieces: the engine, the assets and the rules. He says that rules are the “game logic” or “core design” in the game development. Rules exist as an independent and abstract ingredient of a video game and are neither a part of the engine or the assets. Rather, the rules act as intermediary between the two components, telling the engine how it should govern the assets during gameplay for a specific game.

THORN (2013) claims that assets are all the things a developer must make on a per-game basis. Assets are static and lifeless because, without an engine to guide them, they would be little more than a collection of images and sounds sitting in a folder on the hard drive. Assets include the following: graphics, sound, story, design, animations, scripting, videos, cut scenes, interface components, musical scores and voiceover tracks.

An engine is a framework in which the commonalities of all (or a subset) of games are packaged so that it can be used and reused as the template for building and powering many different games. Generalness and abstractness are the characteristic and essential features of an engine, distinguishing the engine from most other part of games. They are primarily what make it recyclable and powerful and allow it to apply to almost all games (THORN, 2013). The engine also abstracts system components (file and network systems, for example) which deals with the target hardware operational system. Therefore, by using an engine, developers save time that would be spent in coding and debugging certain system and game components, allowing them to focus in programming the gameplay system which increases the overall quality of the game.

GREGORY (2014) claims that a *data-driven architecture* is what differentiates a game engine from a piece of software that is a game but not an engine since the line between game engine and the game is rather blurry. As a simple example, there can be an engine that

has a physics system that the value of its gravity is equal of planet Earth's and cannot be modified. While in another engine you can modify the gravity value to simulate the gravity of other planets. Therefore, he asserts that a "game engine" is a software that is extensible and can be used as the foundations for many different games without major modification.

However, GREGORY (2014) warns that the ideal of a general-purpose piece of software capable of playing virtually any game content has not been achieved yet and may never be. The reason is that most game engines are crafted and fine-tuned to run a particular game on a particular platform or for building games in one particular genre, such as first-person shooters. This limits the level of reusability a game engine has. Gregory explains that specific technologies are employed by engines for some game genres. This phenomenon occurs because designing any efficient piece of software invariably entails making trade-offs, and those trade-offs are based on assumptions about how the software will be used and/or about the target hardware on which it will run. Figure 2 shows all of the major runtime components that make up a typical 3D engine; it shows how complex the structure of a video game can be depending on its scope.

Not all games use engines though. Some games are so simple that the developers do not need to resort to engines. However, by looking at engines the conclusion that can be reached is that there is always will be a part of the software of video games that will not have any bearing with gameplay but it will allow the gameplay to happen in the target hardware.

Video games have an essential component that is not present in the ones that I identified in games. This component is called "event". GREGORY (2014) asserts that video games are inherently event-driven. An *event* is anything of interest that happens during gameplay. Events can be an explosion going off, the player being sighted by an enemy or a health pack getting picked up. Video games generally need a way to notify interested game objects when an event occurs and arrange for those objects to respond to interesting events in various ways, this is called *handling* the event. Different types of game objects will respond in different ways to an event. The way in which a particular type of game object responds to an event is a crucial aspect of its behavior, just as important as how the object's state changes over time in the absence of any external input.

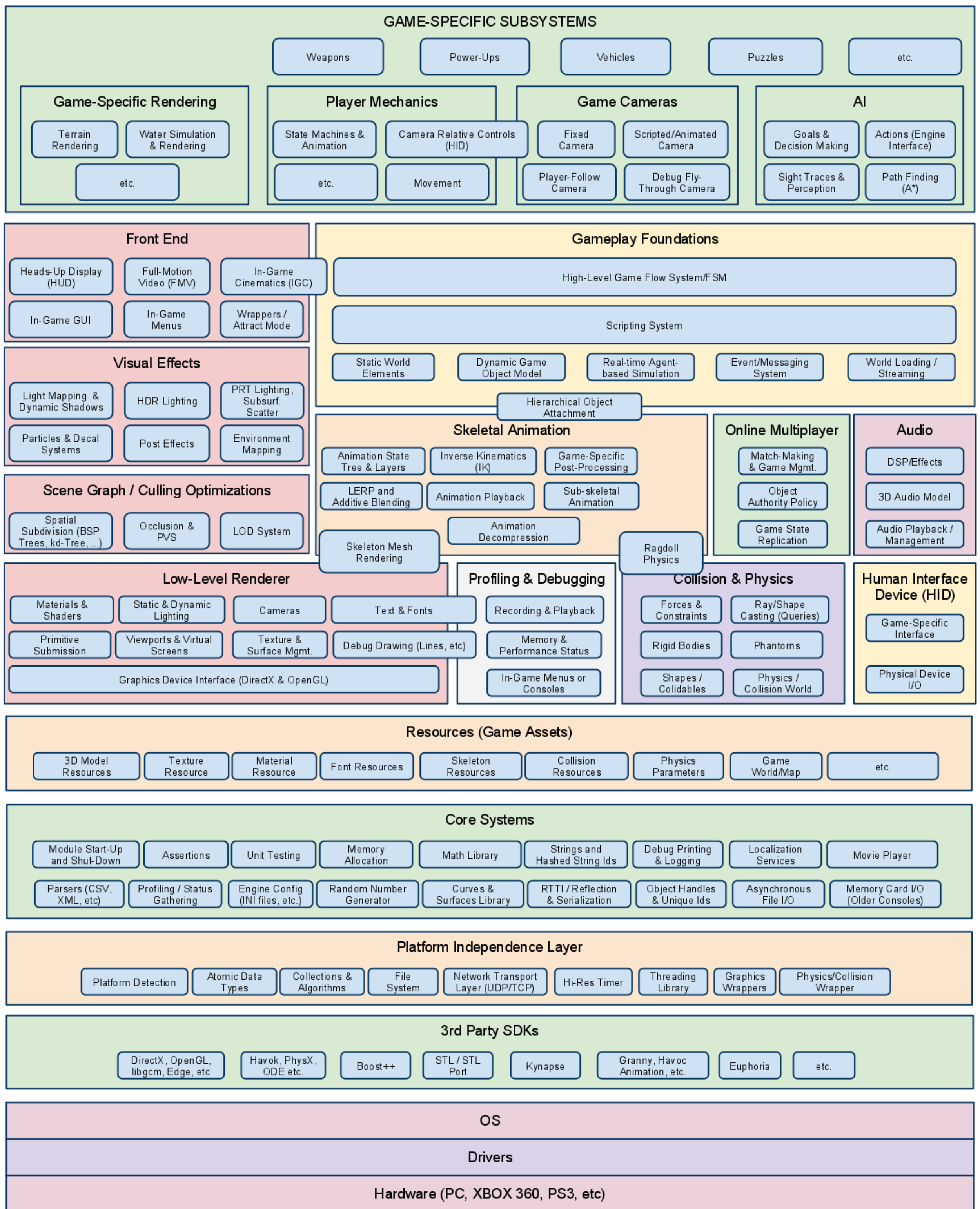

Figure 2 – Runtime game engine architecture (GREGORY, 2014)

Why events are not mentioned in the essential components of games found in the last section? The reason may be the fact that all the *handling* of events in physical games, such as board games, is done by the players. The players are the ones that enforce the rules of the game, they are the ones that change the state of the game and make sure the state is coherent with what they see. Another fact is that some events that need to be configured in video game occur naturally in the real world. An example would be the collision of physical objects, throwing a ball at someone in a video game would require some time implementing it while in the real world you would only need a ball to execute the mechanic. Finally, events can be seen as a subset of rules. For example, "what objects can be hit by a fireball?" shows what objects can respond to the event a fireball hitting them, next is "what happens when a barrel is hit by a fireball?" shows how the barrel objects responds to the event of getting hit by a fireball. Naturally, events can be seen as a subset of the rules of a game but they are an important concept in game programming because game objects invariably need to communicate with one another.

Besides the software part of a video game, there is the hardware part of it and it is equally important as well. The hardware components of video games are the physical devices that receive the player input (keyboards, controllers, microphones) and that provide feedback to the player in form of some output (video, sound, rumble). The platform in which the game runs is the most important physical component but some of them have little bearing in the gameplay. Some platforms, most notably handhelds and mobile phones, have built-in components such as touch-screen displays and microphones. The *Nintendo DS* provides two screens, the bottom one being a touch-screen display. In this case, the platform itself can provide forms of gameplay not available in others.

GREGORY (2014) calls the components that have to process input from players *human interface devices* (HID). Some also allow the software to provide feedback to the human player via various kinds of outputs as well. Most of the inputs fall in the following categories: digital and analog. Digital buttons can only be in one of two states: pressed and not pressed. Analog inputs can take on range of values rather than the two values of a digital input. Also there are special inputs such as chords (multiple buttons pressed together), sequences (button pressed in sequence within a certain time limit) and gestures (sequence of inputs from the buttons, sticks, accelerometers, etc.).

Another important aspect of inputs in video games is that they can be considered external events. Another type of external event would be online data received from sessions from multiplayer games. The software of the game has to process the input from a human player into the appropriate action within the game logic. Thus, we can divide events in external events caused by entities or objects outside the game logic and internal events that are caused by objects within the game logic.

Assets are elements that are used in the output of video games. The final image displayed in the screen to the player is composed of several game elements that are represented by assets. It should be noted that there are rules that govern what elements are displayed according to the game state as well what sound clips are played. What a player sees in the video screen or hear through the sound speakers may not be the asset original form; assets can be manipulated at real-time by the game according to its rules. For example, a 2D image may be stretched or have its colors changed (pallet swaps, common in fighting games). Therefore, it is of great help to categorize game objects according to the outputs associated to them (an object that is displayed, an object that plays sounds) and actions that manipulated the final output to the player (change the color of an object, turn the object invisible). Finally, the output of video game can be considered an essential element since it is composed of objects and can be manipulated by the developers to reach their vision of their game.

Concluding this section, I claim that video games are composed of the following essential elements: assets (images, videos, 3D models), events that are handled by interested objects, inputs sent by an external entity to the game logic, outputs that are composed of one or more objects that can be manipulated and the hardware that allows all that to happen. The software is not included because this is a list of essential components for the game and gameplay design which is independent of the technology used to implement it.

In the next section, I will talk about game design. The development of any type of game begins by its design.

3.3 – Game Design

BETHKE (2003) claims that we have to design a game first and foremost. Game design is the process of creating the content and rules of a game. Good game design is the process of creating goals that a player feels motivated to reach and rules that a player must

follow as he makes meaningful decisions in pursuit of these goals. Good game design is player-centric. That means that above all else, the player and her desires are truly considered (BRATHWAITE & SCHREIBER, 2008).

The focus of a game designer is designing game play, conceiving and designing rules and structure that result in an experience for the player (TEKINBAS & ZIMMERMAN, 2003). Thus, the game designer creates an experience for the player. Games are merely a means to that end because, on their own, games are just clumps of cardboard or bags of bits. Thus, games are worthless unless people play them (SCHELL, 2014). COSTIKYAN (2002) notes that game design is ultimately a process of iterative refinement, continuous adjustment during testing since it is almost impossible to specify a game at the beginning of a project and have it working perfectly.

Game design, as a discipline, requires a focus on games in and of themselves. Because game design is an emerging discipline, it borrows knowledge from other domains – from mathematics and cognitive science; from semiotics and cultural studies (TEKINBAS & ZIMMERMAN, 2003). This makes game design not an easy subject to write about. According to SCHELL (2014), to truly understand game design is to understand an incredibly complex web of creativity, psychology, art, technology, and business. Everything in this web is connected to everything else. Changing one element affects all the others, and the understanding of one element influences the understanding of all others.

SCHELL (2014) asserts that game designers will need all the skills imaginable because almost anything that someone can be good at can become a useful skill for a game designer. He provides a list of them: animation, anthropology, architecture, brainstorming, business, cinematography, communication, creative writing, economics, engineering, games, history, management, mathematics, music, psychology, public speaking, sound design, technical writing and visual arts. There are many more but he claims that **listening** is the most important skill that a game designer can have. It is by listening that the designer will understand the different viewpoints (development team, client, audience, game and self) that participate in the development of a game.

In the next section, I will talk about video game development. I will present the process distinct phases, the structure of a typical game development team and the

responsibilities of each member including of the game designers and the documents produced to support the development process.

3.4 – Video Game Development Process

According to BETHKE (2003), a development process is the method that a developer uses to take the game specifications and turn them into a game. However, learning a process takes time, and most organizations are in short supply of time. They are under great pressure to get something visible and running as quickly as possible to reassure management that the project is well under way. Also, producing digital games is a complex and expensive process. A developer's goal is to produce the best-selling game within the limit of its resources (FULLERTON, 2014). In short, video game development is a problem of time and project management that the developer must solve (THORN 2013).

According to FULLERTON (2014), the industry has evolved standard stages of development to define contracts and milestones for a game project to solve this problem. Best practices for producing games are evolving to recognize the need for flexibility and iteration as part of the game development process. Many developers now use a mix of agile development methods and traditional software production methods to produce their games. The core difference between those distinct development methods is a focus on creating working software versus documentation and managing the team so that it can respond to discoveries in the process, rather than following a predetermined plan (FULLERTON, 2014).

Depending on the size and scope of the game, the development process can be different. For example, AAA games have teams with more than 200 people and a publisher who provides the money for the project as well as handling other business tasks such as marketing. On the other hand, indie games can have a team of 5 people, be financed by the developers own pocket and business task handled by the own developers. It should be noted that because of the electronic games industry competitiveness and corporative way of working, its management and development processes are significant corporate assets and generally inaccessible to the researcher (CALLELE *et al.*, 2005). Thus, the content discussed in this section may not present the state of art employed by the industry.

3.4.1 – Video Game Development Phases

The game development process is composed of many sub-processes and each developer has their own unique methodology of development that can have unique activities. However, any game development can be roughly separated in three general phases: preproduction, production, and postproduction.

3.4.1.1 – Preproduction

BETHKE (2003) claims that a developer should figure out what he has to do before actually doing anything; the game industry term for this phase of work is preproduction, or the vision or design phase in which the developers determine the game contents and scope. He also states that too many projects violate their preproduction phases and move straight to production. In his opinion, preproduction is the most important stage of the project. Finally, it should be noted that because of the iterative nature of video games, a "perfect" project scope will never be achieved, but it is the goal of the manager to develop a solid scope that will help guide the project to its conclusion (KANODE & HADDAD, 2009).

Preproduction entails the conception of a game, identification of the game's requirements, an analysis stage to determine the implications of these requirements, prototyping and the production of design documents detailing game, art, audio, and technical requirements (BETHKE, 2003). By the end of preproduction, the GDD should be finished, though it will be continuously updated during the other phases (KANODE & HADDAD, 2009).

According to FULLERTON (2014), prototyping is the creation of a working model of the developer idea that allows him to test its feasibility and make improvements to it. Also, SCHELL (2014) asserts that the game improves through iteration and he introduces *The Rule of the Loop*: "The more times you test and improve your design, the better your game will be".

Thus, prototyping and iterations are important methods in video game development that can be used to find the elusive "fun" element of a game. Prototyping should primarily be done in the preproduction stage in order to define what the game is and to find the fun element of a game. If this can be accomplished, then production will go smoother. Actions in preproduction determine requirements and affect production (KANODE & HADDAD, 2009).

Requirements engineering should take place at the end of preproduction, once the game designers have found the type of game that is to be created. Gathering all the needed requirements will cut down on the number of iterations needed, and mitigate the late addition of features (feature creep). Once the preproduction phase has been completed, the project manager would take the game design document, and formulate a project scope (KANODE & HADDAD, 2009).

KANODE & HADDAD (2009) conclude that a successful preproduction defines an exciting and absorbing game. Great preproduction reduces the need to find that elusive element of "fun" during the production stage and allows the team to implement the game instead of experimenting it.

3.4.1.2 – Production

Production is the longest and most expensive phase of development. The goal in this stage is to execute on the functional vision established during preproduction (FULLERTON, 2014). Production entails the development of the code that makes the game function, the creation of the diverse types of assets that the game may need to realize the vision of the designers and quality assurance (QA) processes such as testing and debugging.

This phase is the one most fraught with problems. A poor GDD affects project scope which affects production negatively. Feature creep happens at this stage, and can cause delays. A poorly managed production phase results in delays, missed milestones, errors, and defects. In production, the developers often create prototypes, iterations and/or increments of the game. Changes in prototypes or iterations of the game can cause drastic changes to the GDD. Unmanaged changes (or poorly managed ones) can cause widespread problems affecting functionality, scheduling, resources, and more (KANODE & HADDAD, 2009).

In order to mitigate the numerous unknown design problems in game development, agile software development methods were adopted recently by the industry. It is a methodology that strives to make the development more adaptive and people-centric. This means that rather than having developers follow a detailed specification, they will address the priority features to set short-term goals in short periods called "sprints". They meet daily or weekly to evaluate their progress, set new goals, and determine if there are issues halting progress. According to WINGET & SAMPSON (2011), this methodology reduces the

emphasis on pre-documentation in favor of iterative design and dynamic problem-solving guided through frequent team and inter-team meetings.

Testing is usually the last thing done before the game goes “gold” (handed off to the publisher for production and distribution). The testing phase involves stressing the game under play conditions. The testers, not only look for defects, but push the game to the limits (game options set to maximum resolution, textures, etc.).

As it can be seen, the production phase does include planning and prototyping, and other activities that can be found the preproduction and testing phases. Looking at the phases in a broad view, they do seem to fit a waterfall model, though the activities in production break the model with the occurrence of iterations and increments (KANODE AND HADDAD, 2009).

3.4.1.3 – Postproduction

Postproduction involves activities that are done after the game reached “gold” status. It entails distribution, marketing, shipping and maintenance of the game. This phase is largely composed by business activities that are done in order to ensure that the game arrives to the market and make a profit to the developer and the publisher, if there is one.

The amount work done by the developer with maintenance varies with the scope of the game. The most common example is the application of patches that corrects bugs that passed through QA. Another one is the maintenance of online games such as *World of Warcraft*; not only the developer must correct bugs but also provide new content in a timely fashion so that the game stays fresh in order to not lose users.

3.4.2 – Video Game Development Roles

Members of video game development teams include practitioners from such diverse backgrounds as art, music, graphics, human factors, psychology, computer science, and engineering. Individuals who, in other circumstances, would be unlikely to interact with each other on a professional basis unite in their economic goal of creating a commercially successful product (CALLELE *et al.*, 2005).

BETHKE (2003) warns that adding more staff requires more administrative overhead, and there is a critical threshold of number of staff in an area on a project beyond which

impacts negatively on work results, even if the people on the project are competent. This is because the increasingly complex communication required between a large numbers of people on a project as it grows in team size and the different disciplines involved.

Figure 3 shows the separation of the components of a video game in three categories: content, mechanism and technology. The components are separated in sets pertaining to the respective teams that produce them. As it can be seen, there are overlaps between the categories and the sets indicating that some roles will require multiple skill sets to produce satisfactory components.

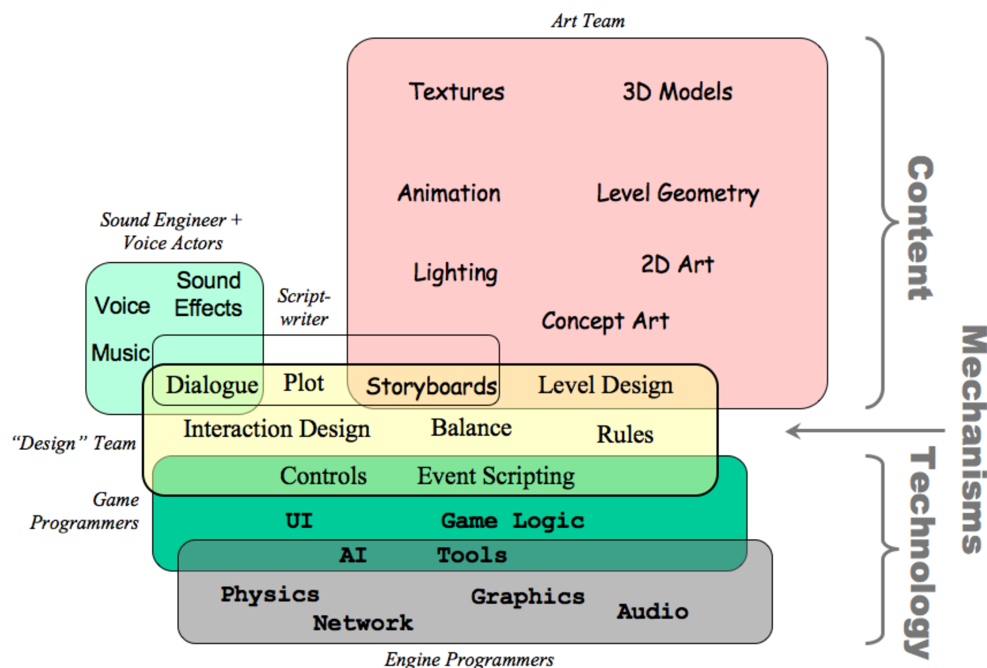

Figure 3 – Decomposition of a game within a development team (LEWIS *et al.*, 2007)

GREGORY (2014) says that game studios are usually composed of five basic disciplines: engineers, artists, game designers, producers, and other management and support staff. Each discipline can be divided into various sub disciplines.

3.4.2.1 – Management and Support

The team of people who directly construct the game is typically supported by a crucial team of support staff. This include the studio's executive management team, the marketing department (or a team that works with an external marketing group), quality assurance team that fix and resolve bugs, administrative staff and the IT department, whose job is to purchase,

install and configure hardware and software for the team and to provide technical support (GREGORY, 2014).

3.4.2.2 – Producers

The simplest definition of a producer for the development team is that he/she is the project leader. The producer is responsible for the delivery of the game to a client or to the market as promised. To make this delivery the producer must create a plan for that delivery, including a schedule, budget, and resource allocation. The producer also need to manage the development team to make sure deliverables are completed on time as well as motivating the team and solving production-related problems (FULLERTON, 2014).

3.4.2.3 – Artists

The artists produce all the visual and audio content in the game. There are many specialties such as: concept artists who produce sketches and paintings that provide a vision of what the final game should look like; 3D modelers who produce the three-dimensional geometry for everything in the game world; texture artists who create the two-dimensional images known as textures which are applied to the surfaces of 3D models; lighting artists who lay out all the light sources in the game world; animators who imbue the characters and objects in the game with motion. Also, there are sprite artists, motion capture actors, sound designers, voice actors, writers, composers and many more. Some game teams have one or more art directors who manage the look of the entire game and ensure consistency across the work of all team members (GREGORY, 2014).

3.4.2.4 – Engineers

GREGORY (2014) says that the engineers are responsible for the design and implementation of the software that makes the game, and tools, work. Engineers are often categorized into two basic groups: *runtime* programmers, who work on the engine and the game itself, and *tools* programmers, who work on the offline tools that allow the rest of the development team to work effectively. On both sides, engineers can have various specialties such as rendering, artificial intelligence, audio, physics, gameplay programming and scripting, etc. There are also the jack of all trades who can jump around and tackle whatever problems might arise during development. Other responsibilities include drafting technical

specifications, making software prototypes, structuring data, managing communications. Documenting code and coordinating with the QA team to fix or resolve bugs (FULLERTON, 2014).

When the engineering team gets big, usually the most experienced engineers are promoted to lead positions. Lead engineers still design and write code, but they also help to manage the team's schedule, make decisions regarding the overall technical direction of the project, and sometimes also directly manage people from a human resources perspective (GREGORY, 2014).

3.4.2.5 – Designers

The job of a game designer is to design the interactive portion of the player's experience, typically known as gameplay (GREGORY, 2014). According to THORN (2013), the most fundamental role of the game designer in the video game development process, whatever his or her motivations for creating a game, is to produce the game design document (GDD). This document must deliver a creative, clear and comprehensive vision of what the game is about and it is supposed to work as a guide or source of orientation for a project. It helps steer the project toward a path to success because members of the team have a common goal or target in mind about how the game is supposed to be or feel. Other responsibilities include brainstorm concepts, creation of prototypes, playtests and revision of prototypes, communicating the vision for the game to the rest of the team, acting as an advocate for the player, etc. (FULLERTON, 2014).

ENGLAND (2014), a game designer at Insomniac Games, talks about all the different types of designers in an effort to help clarify what a game designer does in a video game. According to her, there are designers who dip their hands in all elements of design and sometimes even art and programming, and then there are specialized roles like systems designer, combat designer, level designer and economy designer.

ENGLAND (2014) asserts that any general definition of design has flaws because the actual responsibilities of a designer varies depending on the size of the studio, the platform, the genre, the size of the game, the studio culture regarding roles, how specialized people are, and even whether there is a design department at that studio. The designer on a first person shooter has very different practical responsibilities than the designer on your next favorite

match-three mobile game and, in consequence, needs different skills to fulfill those responsibilities.

Some senior designers have management roles. Many game teams have a game director, whose job is to oversee all aspects of the design, help manage schedules, and ensure that the work of individual designers is consistent across the entire product (GREGORY, 2014).

3.4.3 – Video Game Development Documents

BETHKE (2003) describes the production plan which is a suite of documents that specify the game that is being created. The purpose of creating all of these documents is to know what the development team is going to do. Those documents are:

- Concept/Vision/Proposal Document;
- Game Design Document;
- Art Design Document;
- Technical Design Document;
- Project Plan (set of documents).

It should be noted that some developers can combine those documents in a bigger one or divide them further. As an example, there are several types of documentation commonly available to game artists. The main difference between programming and art documentation is that art documentation is not standardized at all (CONGDON, 2008).

I will talk about the GDD, technical design document (TDD) and project plan which is a set of documents. I will mainly talk about the GDD to show how it influences in the creation of those and other documents.

3.4.3.1 – Game Design Document

THORN (2013) asserts that the central aim of the GDD is to articulate the creative vision of the game in a concise, comprehensive, and technologically independent way to a readership of game developers – usually the other developers on the team. Its content will be tailored more academically to an audience of experienced developers contracted to work on the game project and whose responsibility is to ensure that the design is realized as closely as

possible. Also, the GDD is typically written to be technologically independent, meaning that it will not make directly prescriptive statements or concrete demands about how the game will be implemented (THORN, 2013). Finally, the GDD must be thorough, but not necessarily formal (in the sense of structure or from a mathematical perspective). In fact, one could argue that imposing too much structure on the creative process may be highly detrimental – constraining expression, reducing creativity, and impairing the intangibles that create an enjoyable experience for the customer (CALLELE *et al.*, 2005).

The form of the GDD varies widely across genres and studios. Typically it includes a concept statement and tagline, the genre of the game, the story behind the game, the characters within the game, and the character dialogue. It will also include descriptions of how the game is played, the look, feel, and sound of the game, the levels or missions, the cut scenes, puzzles, animations, special effects, and other elements as required (CALLELE *et al.*, 2005). BETHKE (2003) claims that creating a game design document is so much work that it is natural to break the job up across multiple people to get the work done more rapidly and with higher quality. Because games used to be so much smaller in scope and complexity, it was much simpler to document the game design, so no great amount of formalism was required (BETHKE, 2003).

According to SCHELL (2014), the trouble with GDDs is that they are literally out of date the moment you write them. Design documents are an expression of the designer current theories about what will make the game good but until the designer see those theories in practice, he cannot know. This means that traditional design documentation loses significant accuracy and descriptive ability as development progresses. This is reinforced by WINGET & SAMPSON (2011) interview with a developer who says:

"At some point the document becomes irrelevant because even if we weren't sure how [a game feature] was going to go it becomes faster just to play around with the actual code itself or the editor or the artwork that is in the game and look at in the game and get a feel for it in the game, and then sometimes new ideas come that were never in the GDD to begin with."

Since the GDD communicates what the game should be to the development team. According to BETHKE (2003), this leads to several implications:

- The programming staff must be able to pick up the game design document and efficiently develop the technical requirements and technical design for the software that is needed to be developed for the game;
- The art team led by the art director must be able to read through the game design document and understand the look and feel as well as the scope of the art assets involved in creating the game;
- The game designers on the team must understand what areas of the game require their detailed hand in fleshing out such as 3D levels, dialog, and scripting;
- The audio designers must understand what sound effects, voiceovers, and music need to be created for the game;
- The marketing team should understand what themes and messages they need to use to build the marketing plan around;
- The producers must understand the various components of the game so they are able to break the game down into a production plan;
- The executive management must be able to read through the game design document and be convinced to fund the project.

3.4.3.2 – Technical Design Document

The TDD is the blueprint for the software engineers on the development team to use in the creation of the game. The ideal technical design document will specify to the developers not only what needs to be created but also how it will be implemented (BETHKE, 2003). It must synthesize the requirements of the game, develop a software design, serve as a testing plan, and also supply the project manager with critical information such as the required developer roles, dependencies between tasks and developers, and an estimate of how long it will take to perform each of the tasks assigned to the developers (BETHKE, 2003).

The TDD is formulated from the GDD; this is a troublesome process as it is hard to transform the game designer requirement into technical requirements. CALLELE *et al.* (2011) points that the GDD descriptions exhibit characteristics of a push of information that the preproduction team deems important to the production team. They do not observe similar

evidence that the GDD contains the information that the production team would pull from preproduction because production deems it important. In other words, the traditional GDD appears to serve the producers of the document more than the consumers of the document.

3.4.3.3 – Project Plan

According to BETHKE (2003), the key to successful game development is planning, and you cannot create a good plan without understanding what goals or requirements your plan must fulfill. That is why the GDD and the TDD are essential to make the project goals more defined as you can define tasks and assign those tasks to team members. The project manager will be able to make time estimates for the completion of tasks and the budget to support it. In other words, the project plan can be made.

The project plan contains a schedule that describes what will be accomplished, how long the tasks will take, and who will perform these tasks. It also contains other information such as milestone dates, human resources, task dependencies, legal information and a risk management plan (BETHKE, 2003).

Chapter 4 – Related Work

The objective of this thesis is to develop an ontology that supports the video game development process by assisting in the identification of technical requirements in the game design. Thus, I have researched studies proposing knowledge models about games, video games, game design and video game development. Those knowledge models consist of design guidelines, methods, theories and tools that have been developed over the past years. Some of these were developed specifically to assist the design process, while others were developed as analytical tools, work methods, or documentation techniques (DORMANS, 2012a).

First, I will present studies, made by game design professionals, which propose a vocabulary for games. Those studies are important since ontologies are shared vocabularies and existing vocabularies in the game design domain can be leveraged for the construction of the ontology. Second, I will present informal knowledge models for games. Those models are described in natural language and are not used as software artifacts. Third, I will present formal knowledge models for games. Those models are described in a machine-interpretable language. Fourth, I present ontologies for games as they are different from other formal models because they can be reasoned by the computer. Finally, I will present my conclusion regarding the studies researched.

4.1 – Vocabularies for Game Design

According to CHURCH (1999), game design is the least understood aspect of computer game creation. This is further reinforced as there is no “unified theory of game design”, no simple formula that shows how to make good games as well as a standardized vocabulary for game design (SCHELL, 2014). The lack of a standardized vocabulary is caused by not doing enough to build on past discoveries, share concepts behind successes, and apply lessons learned in one domain or genre to another (CHURCH, 1999). CHURCH asserts that the primary inhibitor of design evolution is the lack of a common design vocabulary and he adds that most professional disciplines have a fairly evolved language for discussion.

TEKINBAS AND ZIMMERMAN (2003) assert that a vocabulary for game design lets game designers talk to each other. It lets them share ideas and knowledge, and in doing so, expands the borders of this emerging field. Media theorist and game scholar Henry

Jenkins identifies four ways that building a critical discourse around games can assist not just game designers, but the field as a whole:

- Training: A common language facilitates the education of game designers, letting them explore the variety and depth of their medium.
- Generational Transfer: Within the field, a disciplinary vocabulary lets game designers and developers pass on skills and knowledge, rather than solving the same problems over and over in isolation.
- Audience-building: In finding a way to speak about them, games can be reviewed, critiqued, and advertised to the public in more sophisticated ways.
- Buffer against criticism: There are many factions that would seek to censor and regulate the content and contexts for gaming, particularly computer and video games. A critical discourse gives us the vocabulary and understanding to defend against these attacks.

CHURCH (1999) claims that a game design vocabulary would allow designers to talk about the underlying components of a game. Instead of just saying, "That was fun," or "I don't know, that wasn't much fun," designers could dissect a game into its components, and attempt to understand how these parts balance and fit together. A precise vocabulary would improve our understanding of game creation and facilitate it. Also, he adds that a vocabulary is a toolkit to pick apart games and take the parts which resonate with the designer to realize his own game vision, or refine how his own games work.

TEKINBAS & ZIMMERMAN (2003) claim that creating a vocabulary requires that designers look at games and the game design process from the ground up, propose methods for the analysis of games, assess what makes a game “fun”, and ask questions about what games are and how they function. The result is a deeper understanding of game design that can lead to genuine innovation in the practice of making games. Part of creating a vocabulary is defining concepts, but this is no simple task, for it involves creating definitions for words that often thread their way through multiple and contradictory contexts (TEKINBAS & ZIMMERMAN, 2003).

One of the main difficulties in establishing a game design vocabulary is that clear statements about game design ideas, and terms referring to them, are introduced all the time (SCHELL, 2014). For example, fighting (*Street Fighter 2* from *CAPCOM)* and role-playing (*Final Fantasy* from *SQUARE-ENIX*) video games have unique concepts and terms when talking about them. For example, fighting games have terms such as a *hit stun* and *chip damage* and RPG games have terms such as *loot* and *buffs*. It could be said that video game genres have their own vocabularies. Another is that the goal of game designers is to provide an experience for the human players, which is an abstract goal with no proven method to reach and the experience of playing a game varies from player to player.

CHURCH (1999) was the first to propose a game design vocabulary in order to improve the game design field in his article *Formal Abstract Design Tools* (FADT) that provides a framework that attempts to provide a shared designed vocabulary and a way to build it. Church breaks down the phrase FADT as follows: "formal," implying precise definition and the ability to explain it to someone else; "abstract," to emphasize the focus on underlying ideas, not specific genre constructs; "design," as in, well, we're designers; and "tools," since they'll form the common vocabulary we want to create. ADAMS & DORMANS (2012) call FADT a design lexicon instead of a design tool because it seemed to be more successful as an analytical tool than a design tool. They assert that this project has been abandoned because it has never caught on.

COSTIKYAN (2002) also attempted to provide a vocabulary for games. He attempts to do so by analyzing various common elements found in various games such as goals, struggle, structure (ecosystem), endogenous meaning, interactivity, entertainment. He also analyzes games by looking at the pleasures that they provide for the players. There are eight: sensation, fantasy, narrative, challenge, fellowship, discovery, expression and masochism.

By reading other books about games and game design from authors such as SCHELL (2014), TEKINBAS & ZIMMERMAN (2003), FULLERTON (2014) and ELIAS *et al.* (2012), I arrive at the conclusion that there are common terms used in the game design field but each game designer provides different meanings and different processes to analyze and implement them in games. For example, SCHELL (2014) provides 108 lenses for a designer to use in the process of designing a game. Those lenses are tools for examining the design of the game as they are small sets of questions a designer should ask about his design.

From this conclusion I arrive at another one: there are several vocabularies about games and game design and they share common terms or elements. Those common elements are surely what make the structure of the game and the designers manipulate those elements in their own ways to make their unique games. An ontology can be built starting using the common elements but I do not recommend using the vocabulary of a game designer as a basis for building an ontology since there is no consensus in the game design domain.

4.2 – Informal Video Game Knowledge Models

In the literature, many authors from academia or professional game designers have proposed informal knowledge models. They provide the techniques, vocabulary, structures and templates to help game designers in specifying and documenting game design details (TANG & HANNEGHAN, 2011). However these are not formalized in a language, they lack key concepts which make them inadequate for video game development and are not machine interpretable. There are several studies that propose such models:

- KREIMEIER (2002) was the first to suggest a design pattern framework but he never actually built one (ADAMS & DORMANS, 2012). BJÖRK *et al.* (2003) proposed *Game Design Patterns*, a description of patterns of interaction relevant to gameplay, it is complemented by a structural framework to describe games in terms of components. Later in their book of same name (BJORK & HOLOPAINEN, 2004) described hundreds of patterns. According to ADAMS & DORMANS (2012) this approach is much more like a design vocabulary than a pattern language because it does not identify common problems and offer generic solution to these problems. Nevertheless, their book is a valuable collection of design knowledge, but it does not tell the reader how to use that knowledge effectively to build better games.
- JÄRVINEN (2003) proposed a typology of rules to better understand rules as a fundamental structure of games. There is also the *400 Project*[10] initiated by designers Noah Falstein and Hal Bardwoord with the intent to collect 400 rules of game design but only 112 rules were listed and the last one was added in 2006.

[10] http://www.finitearts.com/Pages/400page.html

- JUUL (2003) claimed the existence of a classic game model which is a standard model for creating games, a model that appears to have remained constant for several thousand years.
- HUNICK *et al.* (2004) proposed the *Mechanics, Dynamics and Aesthetics* (MDA) framework that described a game as a collection of *Mechanics* to compose *Dynamics* and a collection of *Aesthetics* characteristics of a game. According to DORMANS (2012a), it has been quite influential and it seems to be one of the most frequently recurrent frameworks found in university game design programs all over the world. However, it lacks scrutiny and accuracy.
- JÄRVINEN (2007) introduced "applied ludology", a practical hands-on analysis and design methodology which complements theories of games as systems with psychological theories of cognition and emotion.
- The *Narrative, Entertainment, Simulation and Interaction* (NESI) model proposed by SARINHO & APOLINÁRIO (2008) in order to design the variability aspects of computer games, simplifying the effort of game design projects.

The most interesting study is the *Game Ontology Project* (GOP) proposed by ZAGAL *et al.* (2005). Its primary function is to serve as a framework for exploring research questions related to games and gameplay; it also contributes to a vocabulary for describing, analyzing and critiquing games. The authors developed a game ontology that identifies the important structural elements of games and the relationships between them, organizing them hierarchically. GOP is distinct from design rule and design pattern approaches that offer imperative advice to designers as it does not intend to describe rules for creating good games, but rather to identify the abstract commonalities and differences in design elements across a wide range of concrete examples. The authors have consciously chosen to focus on things that cause, effect and relate to gameplay in order to help characterize and classify the design space of games. BREYER *et al.* (2009) created a reduced ontology for downloadable casual games of the simulation genre using GOP as a base and characteristics of folksonomies for its organization

GOP is not built using ontological engineering techniques and it is not formalized in an ontology representation language which leads to several problems within GOP.

MALCHER *et al.* (2009) in their experiment that uses GOP as a basis to the development of a method of analysis of similar objects applied to the game design process, the participants were students that had no prior knowledge of GOP. They concluded that GOP, when used, was not capable of modelling with precision various elements found in available games because of the difficulty in specifying non ambiguous definitions for a concept and the enormous number of variations in elements of games and their possible uses. The great number of concepts turned out to be troublesome because it increases the time and the difficulty of the analysis process, the concept memorization process and learning the practical use these concepts during the analysis process. This is relevant because it is directly related to the effort a game designer must take to be able to benefit from this method.

TANG & HANNEGHAN (2011) point out that GOP as an ontology struggles to represent the features and constraints of concepts in a machine-interpretable format because it only describes the concepts used in a game from an end-user point of view in explanations with example. This causes the appearance of ambiguous terms since there are similar concepts in GOP and the potential lack of important concepts which happens in GOP. They conclude that it has little use in producing a complete specification of computer game for the purpose of development.

All those problems pointed out by MALCHER *et al.* (2009) and TANG & HANNEGHAN (2011) apply to the informal knowledge models presented. They are designed to be used by game designers or people that are learning about games with the problem that they are an incomplete representation of a game. It should be noted that they are useful for knowledge acquisition since they establish guidelines to identify elements of games and game design. Finally, they cannot be used effectively in the video game development process.

DORMANS (2012a) claims that none of the attempts to provide a game vocabulary has gained enough momentum to become something resembling a standard that spans both industry and academia. These vocabularies require a considerable effort to learn while they are more successful in the analysis of existing games, making them more useful for academics than for developers. In addition, they usually provide a clear theoretical vision on the artifacts they intend to describe.

Finally, COOK (2006) asserts the following: “Academic definitions of game design contain too many words and not enough obvious practical applications where people can actually use the proposed terminology”.

4.3 – Formal Video Game Knowledge Models

In the literature, authors from the academia have proposed formal knowledge models. Despite there being no lack of game design models, there is no model that has surfaced as a standard (DORMANS, 2012b) as they can vary in form and function as well as they cover different aspects of games.

Those models are usually formalized in specific languages and modelling paradigms. An example of a language specifically developed to be a description language for 2D video games is “PyVGDL” (SCHAUL, 2013, 2014). They can also be formalized with mathematics (GRÜNVOGEL, 2005), however it is not a designer-friendly representation of game concepts.

Some authors propose using Model Driven Engineering (MDE) methodology to video game development that use formal models (UML models or ontologies in OWL) for the generation of code (FURTADO & SANTOS, 2006, NELSON & MATEAS, 2007, REYNO & CUBEL, 2008, 2009a) . To support MDE in video game development, REYNO & CUBEL (2009b) created a platform-independent model for video game gameplay specification. Also, models can be used to improve software understanding and error checking in a component-based software architecture (LLANSÓ *et al.*, 2011a) as well as providing a semi-automatic process for moving a class hierarchy to a component-based architecture (LLANSÓ *et al.*, 2011b).

Other authors proposed the use of Petri nets (BROM & ABONYI, 2006, ARAÚJO & ROQUE, 2009) to map games as state machines. According to DORMANS (2012a), it is difficult to capture the essence of game in Petri nets. The number of game states usually is not finite, and their complexity quickly becomes problematic if one tries to model a game in every detail. Also, they are less accessible to game designers.

According to DORMANS (2009), the models that are used to represent game mechanics, such as representations in code, finite state diagrams or Petri nets, are complex and not really accessible for designers. They are also ill-suited to represent games at a

sufficient level of abstraction. He created the *Machinations*[11] framework to represent game mechanics in a way that is accessible, yet retaining the structural features and dynamic behavior of the games they represent. This framework has its own diagrams that act as a domain specific language for a subset of game development.

Machinations diagrams are designed to capture game mechanics. As such, they are not only a design tool; they are also useful as an analytical tool to compare and analyze existing games. This allows designers to observe recurrent patterns across many different games. Also, the syntax of the language is exact as it describes unambiguously how different elements interact. This allowed the development of a *Machinations* software tool, which can be used to simulate and experiment with game systems. It should be noted that the framework focuses on rules and mechanics and does not take into account all elements of game design (DORMANS, 2009).

DORMANS (2009) warns that when one sets out to model anything as complex as games, a model can never do justice to the true complexity of the reality of gameplay. He asserts that the best models succeed in stripping down the complexity of the original by leaving out, or abstracting away, many important details.

4.4 – Game Ontologies

Ontologies formalized in ontology description languages such as OWL offer the advantages of automatic processing and reasoning over its concepts and relations which results in the discovery of implied knowledge and consistency checking. There are several studies regarding ontologies for games:

- LING *et al.* (2007) proposed a use-case based fuzzy ontology constructing methodology for constructing the ontology of an educational game, and encoded the resulting ontology with OWL. Fuzzy ontology can be useful because there are many cases of uncertainty in games. Unfortunately, there is not enough literature on fuzzy ontology used in games and fuzzy ontology is not a very well developed line of research according to the authors.

---

[11] http://www.jorisdormans.nl/machinations/

- CHAN & YUEN (2008) proposed a *Digital Game Ontology* which combines the framework of *Music Ontology* and *Game Ontology Project* concepts, to produce an OWL-based ontology which would be comprehensive enough to process a wide variety of concepts and events of digital games from its media format, its production, and player activities and experience. The ontology was constructed using OWL-DL. Protégé ontology editor and Protégé API were used throughout the development process.
- LING *et al.* (2008) proposed an *Ontology-based Edutainment Development Framework* (OEDF) for educational game development. In this framework the knowledge of game development and the content of textbooks are organized in an ontology, which make it sharable and understandable by computers. The OEDF framework consists of a hierarchy of five layers defined as: Client, Application, Representation and Infrastructure, Service e System. One thing that should be noted is that the framework is technology dependent. The ontology is based on description logic and it is stored in OWL.
- MACHADO *et al.* (2009) demonstrate a trivial use case of ontologies in the domain of games. The authors use OWL as their ontology language of choice, Protégé for ontology design and the *OwlDotNetApi* for integration between .NET and OWL.
- LEÓN & SÁNCHEZ (2010) proposed the construction of an ontology utilizing object-oriented standards through the use of UML and OCL. Their objective was to show that UML was suitable to represent formal models such as ontologies. The created ontology is based on an architecture that produces full motion adventure games for the mobile scenario.
- ROMAN *et al.* (2011) proposed a general method of using RDF and OWL models to represent the knowledge within a fantasy RPG software that helps them analyze possible situations that can come up at any point in the game. The game ontology was created using the Protégé system and the basic elements of the ontology were loosely influenced by the *Game Ontology Project*. To implement the ontology it was used the Jena framework that allows working with RDF, RDFS, OWL, SPARQL and includes a rule-based inference engine.

- CARDENAS (2014) proposed an ontology model for representing any digital educational game regardless of their attributes. The model was developed using the ontology building methodology proposed by NOY & MCGUINNESS (2001).
- RAIES & KHEMAJA (2014) proposed a semantic formalism based on domain ontology for gameplay specification that offers to game designers a precise model to describe, analyze and communicate gameplay from early stages of development of game-based learning systems. The authors use an ontology building methodology for the development of the ontology.

The most interesting study is *Game Content Model: An ontology for Documenting Serious Game Design* (GCM) (TANG & HANNEGHAN, 2011). The authors studied the existing game design models and highlighted the issues with it – incomplete representation of a game and lack of formalism. To address those issues they borrowed concepts from existing game design models to design a completely new model – the *Game Content Model*. They propose incorporating Model-Driven Engineering (MDE) practices into game development because models can be analyzed and translated using transformation engines and generators to synthesize software artifacts consistent with the models automatically.

The GCM, according to the authors, provides a more complete and formalized definition of game design constructs than the existing game design models. It can help novice game design or non-technical domain experts who wish to design computer games or serious game document design specification of a game formally. This model can also be used as a tool to study about the anatomy of a game both from design and software perspective. It can also be extended to include specific concepts that describe components of other game genres. This provides the flexibility for model developers to extend the ontology.

The authors use an ontology development methodology to build their ontology. The development of the ontology follows a bottom-up approach to ensure that each concept introduced in the ontology can be programmatically represented and to encapsulate technical aspects of game development from game designers. It should be noted that the authors do not specify an ontology representation language or a modelling paradigm for the implementation of the ontology. They only specify in their follow-up article (TANG *et al.*, 2013) which describes a *Game Technology Model* (GTM) for use in their *Model Driven Serious Game*

Development Framework. In this framework, the GCM computational-independent models are transformed into a GTM platform-independent model. Those models are specified in XML which is an adequate language for transformations but it does not support very well reasoning processes.

4.5 – Conclusion

The analysis of the literature regarding knowledge models has shown that the several vocabularies proposed by their respective game designers helped in the development of informal knowledge models which in turn helped in the development of formal knowledge models such as ontologies. Each knowledge model uses other models as a basis. This indicates an evolution in the formal representation of game knowledge, even if it is not considered significant. This evolution happens because those vocabularies and knowledge models share common terms and concepts that repeatedly appear.

In summary, the main uses of game knowledge models observed in the literature are: guide game design, game development, analysis of games, game studies, knowledge building, development of another ontology, Model Driven Engineering (code generation).

The analysis of the literature brought up some facts regarding game ontologies:

- Few ontologies adopt ontological engineering methodologies;
- Many ontologies fall short in the formality representation;
- A good amount of ontologies are implemented in OWL (GCM and GOP are not);
- Only GOP is generic. The other ontologies are developed with a certain genre of game in mind;
- None of them are available on the Web;
- None of them use ontologies as a knowledge management solution for video game development;
- None of them are effective in the development of a video game. The most they can do is to be used in prototyping.

Those facts show that the ontological engineering field is not explored enough in video games research because few studies use ontological engineering in the development of

ontologies. However, it seems that the situation is changing as most recent studies use ontological engineering.

None of the ontologies or knowledge models is developed as a knowledge management solution for game development. NIESENHAUS & LOHMANN (2009) present a general framework architecture and implementation examples that show how knowledge management and semantic technologies can be employed to support game development. This article is the only study regarding knowledge management for video game development found in the literature.

Chapter 5 – Building Methodology

In this chapter I will present the building methodology of the ontology. First, I will present the design principles that are going to be followed. Second, I will describe the phases of the methodology and their activities. The methodology will follow the ontology design principles described in Section 2.10 and the methodology proposed in METHONTOLOGY (FERNÁNDEZ-LÓPEZ *et al.*, 1997).

5.1 – METHONTOLOGY

METHONTOLOGY is a methodology for building ontologies either from scratch, reusing other ontologies as they are, or by the process of reengineering them. To improve its applicability it adopted some ideas from the more mature Software Engineering discipline. More concretely, its ontology development process is based on the activities identified in the IEEE standard for software development (FERNÁNDEZ-LÓPEZ *et al.*, 1997).

The METHONTOLOGY framework includes: the identification of the ontology development process, a life cycle based on evolving prototypes and the methodology itself, which specifies the steps for performing each activity, the techniques used, the products of each activity and an ontology evaluation procedure (CORCHO *et al.*, 2006, GAŠEVIC *et al.*, 2009). The development process and life cycle are presented next.

5.1.1 – Ontology Development Process

According to CORCHO *et al.* (2006), the activities in the ontology development process can be classified in three categories as show in Figure 4. The activities are also detailed below but with some differences to Figure 4.

- **Management**: It includes scheduling, control and quality assurance.

Scheduling identifies the tasks to be performed, their arrangement, and the time and resources needed for their completion. *Control* guarantees that scheduled tasks are completed in the manner intended to be performed. *Quality assurance* assures that the quality of each and every product output is satisfactory.

- **Development-oriented (Technical)**: These activities are grouped into pre-development, development and post-development activities.

During pre-development, an *environment study* identifies the problem to be solved with the ontology, the applications where the ontology will be integrated, etc. The *feasibility study* answers questions like: “is it possible to build the ontology?”; “is it suitable to build the ontology?” etc.

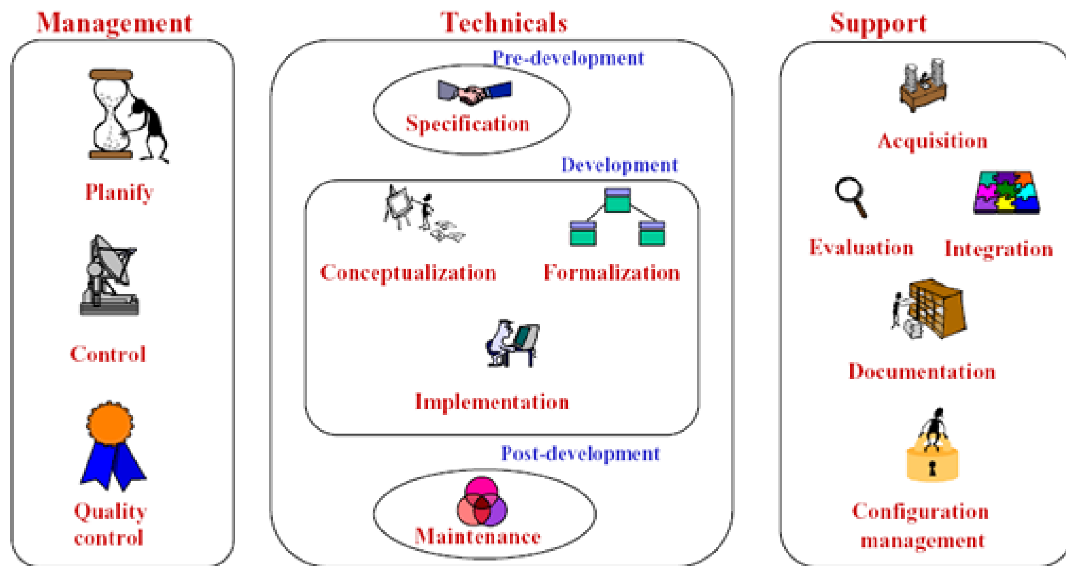

Figure 4 – Ontology Development Process (CORCHO *et al.*,2005)

During development, the *specification* activity states why the ontology is being built, what its intended uses are and who the end-users are, in other words, the scope of the ontology is being defined. The *conceptualization* activity structures the domain knowledge as meaningful models at the knowledge level either from scratch or by reusing existing models. The *formalization* activity transforms the conceptual model into a formal or semi-computable model. The *implementation* activity builds computable models in an ontology language.

During post-development, the *maintenance* activity updates and corrects the ontology if needed. Also, the ontology is *(re)used* by other ontologies or applications. The *evolution* activity consists of managing ontology changes and their effects by creating and maintaining different variants of the ontology, taking into account that they can be used in different ontologies and applications.

- **Support**: These include a series of activities that are performed alongside development-oriented activities, without which the ontology could not be built.

The goal of the *knowledge acquisition* activity is to acquire knowledge from experts in a given domain or through some kind of (semi-)automatic process, which is called ontology learning (Javier Nogueras-Iso et al. 2010). The *evaluation* activity makes a technical judgement of the ontologies, of their associated software environments, and of the documentation. The *integration* activity is required when building a new ontology by reusing other ontologies already available. The *merging* activity consists of obtaining a new ontology starting from several ontologies in the same domain. The *alignment* activity establishes different kinds of mappings between the involved ontologies. The *documentation* activity details, clearly and exhaustively, each and every one of the completed stages and products generated. The *configuration management* activity records all versions of the documentation and of the ontology code to control the changes.

5.1.2 – Ontology Life Cycle

The ontology development process does not identify the order in which the activities should be performed (FERNÁNDEZ-LÓPEZ *et al.*, 1997). This is the role of ontology life cycle, which identifies when the activities should be carried out (CORCHO *et al.*, 2006). Figure 5 shows how the METHONTOLOGY life cycle is.

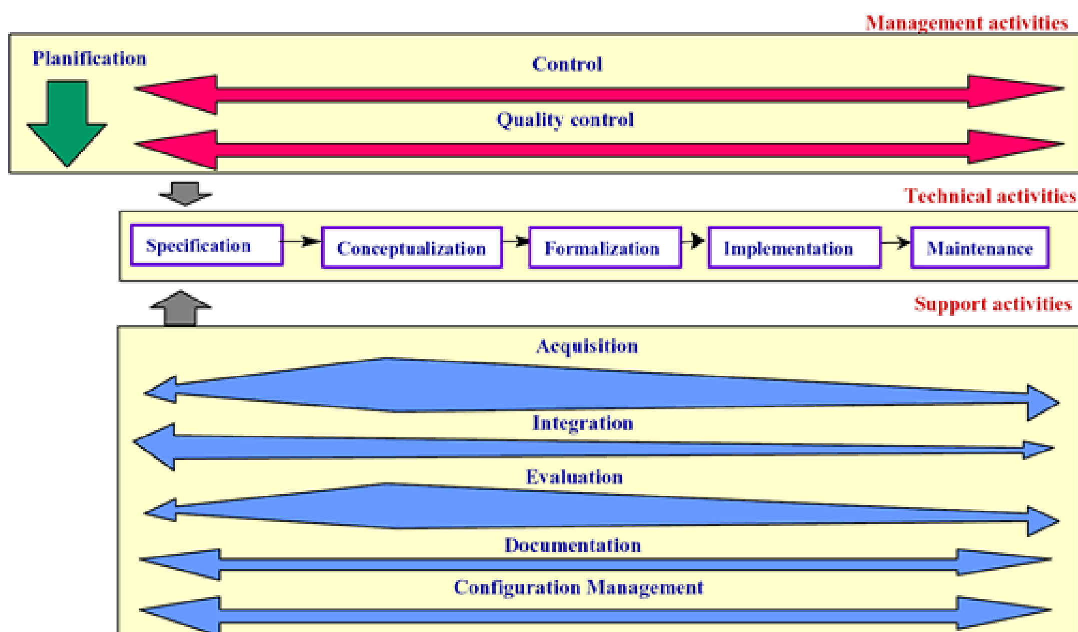

Figure 5 – Ontology life cycle (CORCHO *et al.*, 2005)

The ontology life cycle schedules the ontology development activities detailed previously, **although not all of them are currently considered by the METHONTOLOGY life cycle**. The life cycle is cyclic, based on evolving prototypes (FERNÁNDEZ-LÓPEZ *et al.*, 1997). It allows an incremental development of the ontology that enables earlier validation and readjustment. Each cycle starts with the scheduling activity that identifies the tasks to be performed, their arrangement, their temporal extent and the resources they need. After that the development activities are engaged, starting with specification. Simultaneously, the management activities, control and quality assurance, and the support activities, knowledge acquisition, integration, evaluation, documentation and configuration management, are launched. They take place in parallel with the development activities (GARCÍA-GONZÁLEZ, 2006).

According to GARCÍA-GONZÁLEZ (2006), at each cycle the current prototype ontology moves along the development activities, from specification through conceptualization, formalization and implementation until maintenance, although it is not necessary to pass through all them. Eventually, the prototype might be mature enough for evaluation purposes and a new cycle can be engaged considering the conclusions from this evaluation. He describes the steps performed during a complete development cycle:

- Specification of the prototype ontology;
- Construction of a conceptual model from pieces provided by the knowledge acquisition;
- Formalization of the conceptual model;
- Implementation of the formalized conceptual model. This can be automatic if the formalization can be translated automatically to an ontology implementation language.
- Maintenance of the resulting ontology, which might lead to a new development cycle if unsatisfied or new requirements are detected.

The efforts put into the support activities are not uniform along the life cycle as it is shown in Figure 5. Knowledge acquisition, integration and evaluation are greater during ontology conceptualization. This is because most knowledge is acquired at the beginning of the development, ontologies are integrated at the conceptual level before implementation and

it is better to accurately evaluate the conceptualization as earlier as possible in order to avoid propagating errors (GARCÍA-GONZÁLEZ, 2006).

5.2 – Ontology Building Activities

I will describe the knowledge acquisition, specification, conceptualization, formalization, implementation and evaluation activities in more detail. There will be a chapter dedicated to the specification, conceptualization, implementation and evaluation phases of the ontology building process. Knowledge acquisition was performed in Chapters 3 and 4. The maintenance phase of the ontology will not be covered in this dissertation.

5.2.1 – Knowledge Acquisition

Data collection or knowledge acquisition is a collection-analysis cycle where the result of a required collection is analyzed and this analysis triggers new collections. Experts, books, handbooks, figures, tables, and even other ontologies are sources of knowledge from which the knowledge can be elicited using techniques such as: brainstorming/brainwriting, interviews, observations, document analysis, questionnaires, and data mining (GANDON, 2010).

There is also a set of methods and techniques for the (semi-)automatic processing of knowledge resources. The main aim of this automatic processing, known as Ontology Learning, is to apply the most appropriate methods to transform unstructured (text), semi-structured (HTML pages) and structured data sources (databases) into conceptual structures (ontologies) (NOGUERAS-ISO *et al.*, 2010). The intent is to allow a reduction in the time and effort needed in the ontology development process. A typical prerequisite for enabling (semi-)automated information extraction from Web documents is the use of natural-language-processing and text-processing technologies (CORCHO *et al.*, 2006, GAŠEVIC *et al.*, 2009a).

5.2.2 – Specification

The goal of the specification phase is to produce either an informal, semi-formal or formal ontology specification written in natural language, using a set of intermediate representations or using competency questions, respectively. METHONTOLOGY (FERNÁNDEZ-LÓPEZ *et al.*, 1997) proposes at least the following information to be included:

- The purpose of the ontology, including its intended uses, scenarios of use, end-users, etc.
- Level of formality of the implemented ontology, depending on the formality that will be used to code the terms and their meaning.
- Scope, which includes the set of terms to be represented, its characteristics and granularity.

Any proposal for a new ontology or extension to an ontology must describe the motivating scenario, and the set of intended solutions to the problems presented in the scenario. This is essential to provide a rationale for the objects in an ontology, particularly in cases when there are different objects in different proposals for the same ontology. By providing a scenario, we can understand the motivation for the proposed ontology in terms of its applications (GRÜNINGER & FOX, 1995).

There are three characteristics of the scope of an ontology (GANDON, 2010):

- Exhaustivity: breadth of coverage of the ontology, the extent to which the set of concepts and relations mobilized by the application scenarios are covered by the ontology.
- Specificity: depth of coverage of the ontology, the extent to which specific concept and relation types are precisely identified
- Granularity: level of details of the formal definitions of the notions in the ontology, the extent to which concept and relation types are precisely defined with formal primitives.

One of the ways to determine the scope of the ontology is to sketch a list of questions that a knowledge base based on the ontology should be able to answer, competency questions (GRÜNINGER & FOX, 1995). These questions will serve as the litmus test later: Does the ontology contain enough information to answer these types of questions? Do the answers require a particular level of detail or representation of a particular area? These competency questions are just a sketch and do not need to be exhaustive (NOY & MCGUINNESS, 2001).

These are the informal competency questions, since they are not yet expressed in the formal language of the ontology. By specifying the relationship between the informal

competency questions and the motivating scenario, we give an informal justification for the new or extended ontology in terms of these questions. This also provides an initial evaluation of the new or extended ontology; the evaluation must determine whether the proposed extension is required or whether the competency questions can already be solved by existing ontologies (GRÜNINGER & FOX, 1995).

Ideally, the competency questions should be defined in a stratified manner, with higher level questions requiring the solution of lower level questions. It is not a well-designed ontology if all competency questions have the form of simple lookup queries; there should be questions that use the solutions to such simple queries. Also, every proposal for a new or extended ontology must be accompanied by a set of formal competency questions. It is only in this way that we can evaluate the ontology and claim that it is adequate (GRÜNINGER & FOX, 1995).

5.2.3 – Conceptualization

The objective of this activity is to organize and structure the knowledge acquired during knowledge acquisition using external representations that are independent of the knowledge representation and implementation paradigms in which the ontology will be formalized and implemented next. An informally perceived view of a domain is converted into a semi-formal model using intermediate representations based on tabular and graph notations. These intermediate representations (concept, attribute, relation, axiom and rule) are valuable because they can be understood by domain experts and ontology developers. Therefore, they bridge the gap between people's domain perception and ontology implementation languages (GARCÍA-GONZÁLEZ, 2006).

In order to build a consistent and complete conceptual model, the conceptualization activity defines a set of tasks that should be executed in succession. These tasks increase, step by step, the complexity of the intermediate representations used to build the conceptual model. This way, it is easier to ensure a consistent and complete conceptual model (GARCÍA-GONZÁLEZ, 2006). The conceptualization process of the VGDO will be similar to MEHTONTOLOGY. It is composed of the following steps:

- Rationale behind the creation of the module and natural language description of its properties. There may be sub-sections describing important specialized concepts of the module;
- A diagram that shows the module taxonomy and relations is presented. In some cases because of the module size, there will be a diagram for the taxonomy and one for the relations of the module;
- A concept table to describe the concepts name, their parent concept, which concepts they are disjoint with and membership conditions necessary for a concept to be specialized;
- An attributes table to describe instance and class attributes of the concept. Class attributes have the same value for all instances of a concept, while instance attributes have different values for each instance of the concept. It should be noted that in the case of this ontology, the attributes described in this table are not related to the *Attribute* element. Also, all instances of the concepts of the ontology have a name and description attributes, those are self-explanatory;
- A relations table that describes the binary relations of the elements of the module with elements of itself and other modules. Relations are determined by their name and the source and target concepts. For each relation, its cardinality, inverse relations and mathematical properties (symmetric, transitive, functional, etc.), are specified if possible;
- An axiom or rules table of the module is presented. They are used for constraint checking and for inferring values for attributes and concepts.

5.2.4 – Formalization and Implementation

The goal of the formalization activity is to transform the conceptual model into a formal or semi-computable model. According to CORCHO *et al.* (2003), when formalizing an ontology, it is important to find a formalism which provides adequate primitives to capture the aspects of the ontology. Knowledge representation languages can be used for ontology formalization. An overview of knowledge representation languages is presented by CORCHO *et al.* (2003). Examples of such languages include Classical Propositional Logic; First-Order

Logic; Semantic Networks; Conceptual Graphs; Frames; and Description Logics (BRACHMAN & LEVESQUE, 2004).

The goal of the implementation activity is to build computable models using ontology implementation languages. If this formal language is standardized and if there are platforms complying with the standard, then at least a minimum set of operation is certified and the computational commitment exists (GANDON, 2010). There are ontology development tools that automatically implement the conceptual model into several ontology languages using translators. Therefore, formalization is not a mandatory activity (GARCÍA-GONZÁLEZ, 2006) because it is automatically done in the implementation phase.

5.2.5 – Evaluation

According to GARCÍA-GONZÁLEZ (2006), the evaluation activity judges the developed ontologies, software and documentation against a frame of reference. Ontologies should be evaluated before they are used or reused. There are two kinds of evaluation, the technical one, which is carried out by developers, and user's evaluation.

Ontology evaluation includes the following activities (GÓMEZ-PÉREZ *et al.*, 1995):

- Ontology verification refers to building the ontology correctly, that is, ensuring that its definitions implement correctly the requirements or function correctly in the real world.
- Ontology validation refers to whether the ontology definitions really model the real world for which the ontology was created.
- Ontology assessment is focused on judging the ontology from the user's point of view. Different types of users and applications require different means of assessing an ontology.

The criteria for ontology evaluation are (GARCÍA-GONZÁLEZ, 2006):

- Consistency, which checks if all individual definitions (axioms) are consistent and if no contradictory knowledge can be inferred from other definitions (axioms). Some consistency problems are: circular definitions, common classes or instances in disjoint decompositions and partitions, external instances in exhaustive decompositions and partitions and semantic errors.

- Completeness. All that is supposed to be in the ontology is explicitly stated in it, or it can be inferred. Some common completeness errors are: incomplete concept classification, disjoint knowledge omission and exhaustive knowledge omission.
- Conciseness. An ontology is concise if it does not include unnecessary definitions, explicit redundancies between definitions do not exist and redundancies cannot be inferred. Some redundancies are: redundant subclass of or instance of relations and identical formal definitions of classes or instances.

Another form to evaluate the ontology is to check if it can answer the competency questions defined in the specification phase. The axioms in the ontology must be necessary and sufficient to express the competency questions and to characterize their solutions; without the axioms we cannot express the question or its solution, and with the axioms we can express the question and its solutions. Further, any solution to a competency question must be entailed by or consistent with the axioms in the ontology alone. If the proposed axioms are insufficient to represent the formal competency questions and characterize the solutions to the questions, then additional objects or axioms must be added to the ontology until it is sufficient (GRÜNINGER & FOX, 1995).

Chapter 6 – Specification

According to GRÜNINGER & FOX (1995), any proposal for a new ontology or extension to an ontology must describe the motivating scenario, and the set of intended solutions to the problems presented in the scenario. This is essential to provide rationale for the objects in an ontology, particularly in cases when there are different objects in different proposals for it. By providing a scenario, we can understand the motivation for the proposed ontology in terms of its applications.

In this chapter I will present the specification of the Video Game Development Ontology (VGDO). First, the motivating scenario and the purpose of the ontology will be presented, as they were in Chapter 1. Second, I will describe its intended uses and users. Third, I will detail its characteristics. Fourth, I will determine its formality. Fifth, I will present the sources of knowledge used in the construction of the ontology. Finally, I will present the ontology scope and its main elements.

6.1 – Purpose

There are several problems that exist in video game development. For example, PETRILLO *et al.*'s (2008) survey shows that all the main problems of traditional software industry are also found in the games industry. There is a gap of communication between the members of the development team because it is composed of professionals of different domains of knowledge. This diversity of knowledge and the resulting communication gap reflects negatively on the production of documentation and the requirements identification process because of the lack of standards and common vocabulary in the development team.

The ontology has two purposes: the first is to bridge the gap of communication between the programming and design teams by providing a common vocabulary. The second is to assist in the transition between preproduction and production phases of the game development process by assisting in the identification of implied knowledge (which would be all kinds of requirements) in GDDs. By achieving those purposes, the ontology makes the gathering of requirements more accurate and reliable and, in consequence, mitigates several problems in the game development process.

To achieve those purposes the ontology describes the video game in development or parts of it in a set of accurate and unambiguous terms, those terms can be decomposed in atomic terms that serve as the base of the ontology. This is necessary because using a common vocabulary can be bad for the communication if its terms are inaccurate and not intuitive for the intended audience. A confusing and poorly thought vocabulary is of little use.

It also can be used as a knowledge repository by the development team to keep track of design changes and make queries to clarify concepts and the relationships between them. Because of the formalism that the ontology relies on, automatic reasoning on them can be performed to identify implicit knowledge and requirements. This can help designers and other team members to find flaws in the game design, necessary assets, patterns, etc.

6.2 – Intended Users

The designers design the gameplay of the game as well as its other aspects and the programmers implement the finished designs and integrate all assets with the code to bring to life the experience intended by the designer. There are other teams that participate in the development process but the game is born from its design; from the design the necessary requirements for the production of the game are identified; assets (art, music) are produced from the requirements; finally, all of those assets are integrated with the game logic that is implemented into the software. Thus, the intended users of the ontology are the design and programming teams since all other teams heavily depend on the design of the game and the programming team is tasked with the implementation of the design as well as identifying technical requirements that affect the other teams by stating what is and what is not possible to be done with the available technology. Finally, it is vital that both teams communicate clearly for the production of a quality product.

6.3 – Characteristics

The ontology must satisfy the criteria for appropriate knowledge management solutions for video game development, which were proposed by NIESENHAUS & LOHMANN (2009) and described in Section 1.1.7.

6.4 – Formality

Formality is important for two reasons: first, the ontology has to be machine-readable. Second, the ontology must be unambiguous, with a precise (mathematical) meaning. Such formality helps programmers in the identification of requirements and allows reasoning of the ontology to reveal implicit knowledge and inconsistencies in the design of the game.

The ontology will be rigorously formal as described in Section 2.8 because the ontology will need to enforce constraints such as rules of games that include time limits, space limits, entities that can only interact with certain types of entities, conditions for actions to be performed, etc.

In order to achieve this level of formality, the ontology will be formalized and implemented in OWL 2 using the Protégé ontology editor. OWL 2 is popular, widely used by many people, supported by many applications and it has substantial documentation (tutorials, examples, etc.). It was chosen because of my unfamiliarity with ontology representation languages and the lack of time to learn and compare other ontology languages. Therefore, OWL 2 is the safest choice because if technical difficulties arise during the ontology implementation there will be plenty of resources to consult.

The Protégé editor was chosen because it is open source, has substantial documentation and has already been through a number of versions and modifications. Protégé supports several ontology representation languages, including OWL and RDF(S). Some forms of reasoning over ontologies developed with Protégé are also facilitated; for example, since OWL is based on description logics, inferences such as satisfiability and subsumption tests are automatically enabled. Protégé's plug-in-based extensible architecture allows integration with a number of other tools, applications, knowledge bases, and storage formats (GAŠEVIC *et al.*, 2009b).

6.5 – Knowledge Sources

Knowledge sources are documents or persons which knowledge about the domain is elicited. There are two groups of knowledge sources that were analyzed in Chapters 3 and 4.

The first group of sources of knowledge was academic papers, online articles, books and blogs about games, video game development, video game programming and game design. The majority of authors is of experienced professionals in the game industry or had some

experience designing games. The reader is referred to the bibliography of this dissertation to check the references I used to develop the ontology.

The second group of sources of knowledge was game ontologies. The most notable ontologies are the *Game Ontology Project* (GOP) proposed by ZAGAL *et al.* (2005) and the *Game Content Model: An ontology for Documenting Serious Game Design* (GCM) proposed by TANG & HANNEGHAN (2011).

Those groups were limited to the games, video games, video game development, video game programming and game design domains in order to obtain a consistent vocabulary that can be easily adopted by video game development teams.

6.6 – Scope

According to Section 6.3, the ontology is supposed to be generic and adaptable. This means that the ontology must be able to describe all or part of the elements that compose a video game regardless of genre and technology. It also must give the developer the freedom to abstract or detail a concept as he sees fit. Another characteristic is that the ontology is lightweight as described in Section 2.8 and combined with the fact that the intended users are designers and programmers of the development team; it means that the terms of the ontology must be easily understood by the intended users.

The tendency is that both programmers and designers relate to the top terms of the ontology because it is meant as a vocabulary to be used by both of them. As the terms become more specific they can either approach the game designer or the programmer domain. For example, the terms from the game designer domain are by nature more ambiguous and inaccurate but they will become the opposite because they will extend the top elements of the ontology which are meant to be accurate and unambiguous. The intention is to not use terms specific to programmers and game designers in the construction of this ontology because it is meant to be technology independent and to not describe game designer techniques that are used to balance the game and build an experience for the player. However, the ontology can be extended to include terms more specific to programmers and game designers if the user necessitates. For example, the attributes of a game object can be more detailed as to include datatypes and operations that can be used to modify them. This would be something extremely useful to programmers.

Although I claim that the ontology is to be technology independent, some technology-related terms might appear such as input hardware. This fact is supported by CRAWFORD's (1984) claim that the game designer who designs computer games must thoroughly understand the medium with which he works as the computer offers special possibilities and imposes special constraints (such as limited input) on the designer. It helps in the development process if the designer is aware of the limitations of the available technology.

The ontology is going to be composed by elements that were chosen by me after reading the available literature on game design, video game development and game ontologies. The ontology will be separated in two connected parts: internal and external. Each of those parts will be composed of modules. Internal modules represent the elements located exclusively within the digital game world (rules, objects, actions, etc.) and the external modules represent the elements outside of the game world and the ones responsible for the communication between the internal and external objects.

Each module will be treated as a smaller ontology as their purpose, scope, elements, relations and competency questions are detailed. Also, every element of the ontology has title (or name) and description properties. Both are self-explanatory but I add that the description is important because the developer can describe the element in natural language in case he cannot or has difficulty to describe the concept with the ontology terms.

The internal modules are based on the essential elements that games are composed of, which I identified in Section 3.1.1. The *Game Object* (GO) module will be the root module of this part of the ontology and more specific modules will be born from those. Each module in isolation can be used to describe basic elements within the game world. Moreover, the modules have established relationships to other modules; it is useful to be able to break down the game in distinct parts and observe how these parts interact with each other. Modularization will also facilitate the ontology evaluation, maintenance and extension. The internal modules are:

- Game Object: Describes the objects that are part of the game, the relationships between them, their action, their attributes, the events they handle and their possible states;

- Attribute: A GO can have several attributes such as numbers and strings. This module describes the types of attributes and their range of values;
- Event: it brings information not available internally for a GO (in other words, information about other GO or external objects). The GO takes an Action according to its state and the information received from the Event. This module describes the possible types of Events, the states that trigger their creation, their attributes and the actions that they may cause to happen;
- Action: it changes the state (attributes) of the game and is associated to GOs. The Action changes the GO state, possibly triggering new Events. This module describes how Actions may be structured, the conditions necessary for its execution and the changes they make once executed;
- State: it represents the current state of affairs of an object, be it in the game world or the real world. A State can be defined by the values of the attributes of a GO. States can have sub-states and lead to other states depending on the changes in the attributes of the present objects. This module describes the different types of states, the attributes values that define them and the objects that they belong;
- Space: Gameplay happens in some kind of space. It describes the different types of spaces that can exist in a video game;
- Time: Gameplay happens in a frame of time. It describes the different types of time that can exist in a video game.

The external modules are based on the essential elements that video games are composed of, which I identified in Section 3.2.1. The *External Object* (EO) module will be the root module of this part of the ontology. All external modules extend in some way the internal modules of the ontology with the exception of the EO module. While a game can be described with the basic modules, they are too generic to be used by video game developers as they lack important terms related to video games. With the external modules the development team will be able to describe a video game with more precision:

- External Object: It consists of objects outside the game world that are necessary for the video game to provide the intended experience. They consist of hardware, software

(including other video game) and players that trade input and output between them. EOs trade inputs and outputs with GOs. It can be extended to *Hardware*, *Software* and *Player* modules;

- Software: Describes the software that the video game receives input from and sends output to. Most of the times, those software include operating systems, clients applications (*Steam* for example), servers, etc.
- Video Game: Extension of *Software*. It describes general information about the video game in development such as name, developer, genre and platform. A game is composed of modules; be it one or many. A game can also have other games inside or be a collection of games, so each contained game is a module itself. The modules are composed of several GOs;
- Player: Describes the possible interactions the player can have with the game and the objects manipulated by the player;
- Hardware: The input and output cannot exist without an associated hardware (a controller will hold some buttons and it has the actual state of the inputs). Describe the hardware objects responsible of sending input to the games and displaying the output of the game logic;
- Input: Inputs are sent by an EO to the video game. They are treated as external events. It can be divided in further modules;
- Output: The game sends output to the player. The state of the game and its changes are outputted to a hardware which will transform it in a form discernible to the player. It can be divided in further modules;
- Video Output: Extension of Output. This extension will be detailed as it is the defining feature of a video game;
- Assets: They are artifacts that are used in the output of video games. They usually are images, video, music and other types of files that are integrated into the game code.

As it can be seen, there are no rules and goals modules even though they were identified as essential elements in games. According to SCHELL (2014), rules are the most fundamental elements of the game because they define the space, the timing, the objects, the

actions, the consequence of the actions, the constraints on the actions, and the goals. As it can be seen, rules encompass a lot of distinct concepts that were separated in the above modules and through them we can ask about game rules such as an action that is allowed in a determined game state or the duration of a certain game state.

Goals are created by game designer to guide the player behavior towards the completion of the game; obviously this information is of no use to programmers. Therefore, goals would be a module that would be located on the game designer and player knowledge domain, what the ontology needs are modules that are located in both the game designer and programmers knowledge domain. However, non-player characters (NPC) in some games are programmed with artificial intelligence (AI); the actions of such characters are guided by formal rules or goals. Moreover, NPC can share the same goals of human players. For example, in most fighting games both players (human and non-human) have as objective depleting the opponent`s life to zero in order to win a round. Other concern is whether some NPCs can be considered to have an AI or goals because of the simple behavior, take for example the *Goombas* of *Super Mario Bros.* which only move side to side but they are "smart" enough to not fall from the platform they are on. As it can be seen, the discussion of goals is a complex topic but it is worth considering as a more advanced module once the initial modules are developed and tested.

The terms utilized in the ontology are to be shared by the knowledge domain of game designers and programmers and easily understood in their respective domains. Obviously, more specific terms that are far from those knowledge domains can be created. With the base of the ontology being composed of accurate and unambiguous terms, more specific terms can be represented appropriately for programmers and game designers. It is valid to reiterate that this ontology is for the description of a video game or parts (or aspects) of it in order to assist the development process; it is not adequate for the representation of game designer techniques, representation of physical games, game studies, player behavior, etc.

I created a list of competency questions to better determine the scope of the ontology. These questions will also serve as a tool to evaluate the ontology. They are located in Appendix C because it is a very extensive list.

To conclude, GRÜNVOGEL (2005) asserts that, in general, there is no model of a game capable of representing every aspect thereof, since it is a model and has to leave out certain aspects of the game. SCHELL (2014) states that is up to the developer to decide which objects have certain attributes and certain states. There are often multiple ways to represent the same thing. In a game of poker, a player's hand can be defined as an area of the game space that has five cards objects in it, or defined as an object that has five different card attributes. As with everything in game design, the right way to think about something is whichever is the most useful at the moment (SCHELL, 2014).

The same applies to ontologies too, as they describe part of the knowledge domain. It is impossible for ontologies to describe the totality of the knowledge domain or to accurately describe certain concepts, even though some aim for it. The reason for that is that knowledge is always being created and added to a knowledge domain, thus the ontology will always be incomplete because, for example, it may not describe certain properties of an object since they are not of use to the intended users of the ontology. It is crucial that it is understood that an ontology is by no means the only way to describe a knowledge domain as there are different viewpoints that the domain can be seen from and, in consequence, there might be missing important knowledge from the domain.

Chapter 7 – Conceptualization of Internal Modules

The objective of this chapter is to organize and structure the knowledge acquired during the specification. As it has been presented in the previous chapter, the video game development domain is a very complex one and conceptualizing it is a very challenging task. The process of conceptualization will follow the one presented in Section 5.2.3. As stated in Section 6.6 (scope of the ontology), each of the main elements will be treated as a separate module. For each module, the conceptualization will be done following the same steps.

In this chapter, the internal modules of the Video Game Development Ontology (VGDO) will be conceptualized and detailed. The external modules of the ontology will be conceptualized in the next chapter. The conceptualization process will be the same for the external modules. The concepts, attributes, relations and axioms tables of the modules are presented in Appendix B.

7.1 – Game Object Module

This is the root module of the internal part of the ontology. It is from this module that other internal modules will be born. There are five foundations that help determine a game object's most essential properties and relations. Because of this, all internal modules are related to each other.

The first foundation is SCHELL's (2014) and GREGORY's (2014) definitions of game objects (GO). SCHELL (2014) claims that a game space will surely have objects on it. Anything that the player sees or manipulates (characters, menu) in a game are GOs. GREGORY (2014) is more specific by stating that a GO refers to virtually any dynamic element (an element that changes its state over time) within a game world. Static elements are the inverse since they include pretty much anything that does not move or interact with gameplay in an active way. However, he states that the term GO is by no means standard within the industry. GO are commonly referred to as entities, actors, agents, etc.

GREGORY (2014) states that a GO is essentially a collection of attributes (the current state of the object) and behaviors (how the state changes over time and in response to events). They are classified by type and different types of objects have different attributes schemas and different behaviors. All instances of a particular type share the same attribute schema and the same set of behaviors, but the values of the attributes differ from instance to instance.

The second foundation is CHANDRASEKARAN *et al.*'s (1999) claim that although differences exist within ontologies, general agreement exists between ontologies on many issues. Those properties are:

- There are objects in the world;
- Objects have properties or attributes that can take values;
- Objects can exist in various relations with each other;
- Properties and relations can change over time;
- There are events that occur at different time instants;
- There are processes in which objects participate and that occur over time;
- The world and its objects can be in different states;
- Events can cause other events or states as effects;
- Objects can have parts.

The third foundation is GREGORY's (2014) claim that games are inherently event-driven. According to him, an event is anything of interest that happens during gameplay. Different types of GOs will respond in different ways to an event. Finally, he says that most objects in a game do not need to respond to every possible event. Most types of GOs have a relatively small set of events in which they are "interested".

The fourth foundation is that GOs perform actions to change the game state. While SCHELL (2014) and BJÖRK & HOLOPAINEN (2003) state that players can only change the game state by performing actions, they only consider actions as something that only players do. However in a video game the actions of a player are limited by the interface he is using, many actions that would be done by the player such as moving a piece of chess or calculating the damage of an attack in an RPG are done by the underlying software. Also, there are non-player characters (NPC) with their own AI that tries to replicate a human opponent, so their actions need to be designed.

The fifth foundation is that GOs have distinct states determined by the current values of their actual attributes. According to Jesper Juul, in a lecture titled "Play Time, Event Time, Themability", a game is actually what computer science describes as a state machine.

However, games can exist in a vast number of states making it impossible to document them all. The same applies to GOs; they can have an unlimited number of states.

Therefore, the purpose of this module is to describe the GOs by detailing the relations between different objects, the attributes of an object, the states that a game object has, the events that it reacts to and the actions it performs to change the object or the game state. This module is solely composed of the GO element and has relations with all internal modules. Figure 6 presents a diagram with a simplification of this module which has the following properties:

- A GO can contain other GO elements. If the GO can contain more than one instance of a type of GO, the GO will have a Collection of the type of GO. A common case in an RPG inventory where items with different properties and effects are stored in the same space. **See Axiom 1 at Appendix B**.
- A GO is composed of other GOs. A “composed of” relation is different from a “contain” relation because in the former the GO must have an instance of one while in the latter it is not necessary. For example, a car must have wheels to be considered one while having fuel is not necessary for it to be considered. However, the car engine needs fuel in order to make the car move. The car can contain fuel and different amounts of fuel determine the car behavior.
- A GO handles and sends *Events*. An example is a life bar decreasing when a character takes damage in an action game. The *Event* here is the change on the Life attribute of the character and the subsequent *Action* is the modification of the life bar to reflect the change.
- A GO performs *Actions*. The GO reacts to an *Event* by performing none or many *Actions*. Continuing from the *Event* example, the Action here is the modification of life bar length to indicate to the player how much damage was done.
- A GO has one or many *States* that reflect the current values of its attributes.
- A GO can have one to many reactions to an *Event*; each reaction may be only possible in certain *States* of the GO.

- A GO has *Attributes*. Examples are an avatar position in a 2D space or the amount of life a character has in a fighting game.
- There are types of Game Object. A type can be defined by either the attributes or the events or the actions or a combination of the three. An example of Game Object type would be an object that can move in a 2D space, an instance of this would be Mario in *Super Mario Bros*.

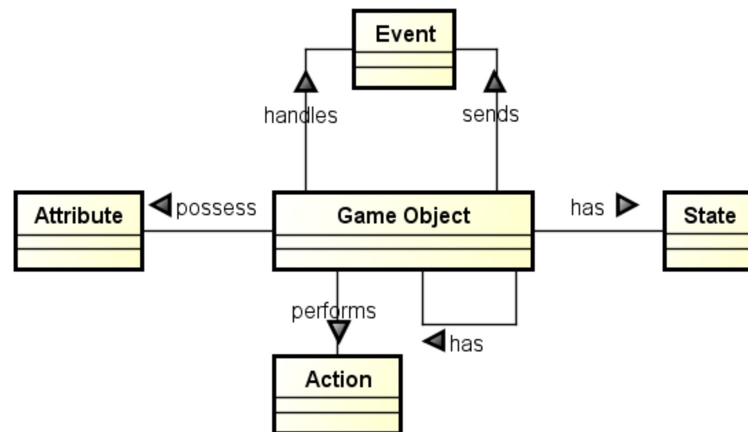

Figure 6 – Game Object Module

7.2 – Attribute Module

According to SCHELL (2014), attributes are categories of information about an object. He gives the example of a racing game where a car might have maximum speed and current speed as attributes. Each attribute has a current state. The state of the "maximum speed" attribute might be 150 mph, while the state of the "current speed" attribute might be 75 mph if that is how fast the car is going. Maximum speed is not a state that will change much, unless perhaps the player upgrades the engine of the car. Current speed, on the other hand, changes constantly.

The reason for a module for attributes is because one single attribute can influence the behavior of several components of video games. A good example is life or health points. There are multiple examples: in the *Legend of Zelda* series when the player is low on health a beeping sound keeps playing until he recovers health to a certain level. In the case of *Wind Waker*, *Link* will look tired and the beeping sound will play. Also, health points are represented as hearts to the player. When health is lost, the hearts become empty. And obviously, when *Link* health reaches zero it is game over.

Therefore, the purpose of this module is to describe *Attributes* by detailing their types, constraints, actions that change them and events triggered by them. First, I will describe the five types of Attribute elements:

- An *Atomic Attribute* is simply composed of the value and datatype it possesses. It includes numbers, strings and boolean attributes.
- A *Composite Attribute* can only be composed by *Atomic* or *Composite Attribute* elements. A *Composite Attribute* does not have a value property but it can have custom properties that can be derived from the values of the *Attributes* it possesses. For example, the *Position* attribute in a 2D space is composed of the X and Y coordinates that are numbers. As for a custom property, a *Vector* has a size property which is calculated using the values of the X and Y coordinates. **See Axiom 2a**.
- *Collection*: It includes structures used for storing a quantity of GOs or *Attributes* such as maps and array lists. Typical operations are insertion, removal, copying and modification of a particular item in a collection. A more complex operation would be the ordering of the items following certain criteria.
- *File*: Assets of a game are files. *Files* have particular operations which transform them in a format such that the game logic can understand them. Typical operations are opening, reading and closing files. More specific types of files such as music and image have their own unique operations.

Figure 7 presents a diagram with a simplification of this module which has the following properties:

- An *Attribute* is present in one or many GOs. For example, objects that move in 2D spaces all have a *Position* attribute that contains the object coordinates. But an instance of an Attribute belongs to only one instance of a GO.
- An *Attribute* can have a range of permitted values. In RPGs there are some examples such as an upper limit to the maximum health points (HP) a character can have or a maximum number of characters the name of the player's character can have.
- An *Attribute* value is changed by an *Action*. Depending on the type of the *Attribute*, it can be changed in a number of ways. For example, numeric *Attributes* can be changed

by adding it to or subtracting it from another number. Addition and subtraction can be categorized as *Action* elements.

- An *Attribute* can only be changed by *Actions* of the same GO. **See Axiom 2**.
- An *Attribute* value or range of values determines the *State* of the GO. For example, in an RPG when the character magic points (MP) reach a certain value, he will not be able to use spells or skills because the MP is too low to pay the cost. Here, we have two states called "Character cannot use magic" and "Character can use magic" which are defined by the MP value.
- It is implied that an *Attribute* value or a combination of values of different *Attributes* can trigger an *Event* because *States* trigger *Events*. For example, when the player's life reaches zero points in an action game, generally it triggers the Game Over event. However, in certain games, if the player possesses a certain GO, instead of receiving a Game Over screen, the character's life is replenished but the GO is lost in the process. In this example, the life attribute is part of the trigger of the event together with the GO. **See Axiom 3**.

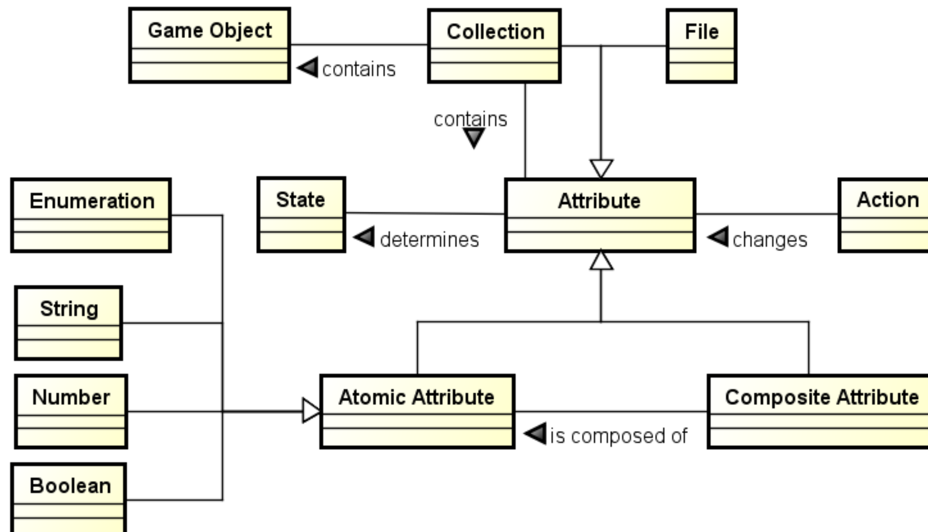

Figure 7 – Attribute Module

An abstract representation of a type of attribute can be modelled with the *Attribute* module. However, the values, constraints, actions and event triggers are unique to an attribute type. For example, the addition operation of a number cannot be performed in a piece of text or in a custom attribute type.

It should be noted that *Composite Attributes* can be present in important parts of the game. For example, the *Vector* attribute is a *Composite Attribute* that has attributes such as length and direction that are calculated through operations. It also has its own addition and subtraction operations. An example of heavy use of vectors is 3D gaming; all game objects present in a 3D space have at least one vector. In the next section, I will describe the most basic attribute types that are present in *Atomic Attributes*.

7.2.1 – Atomic Attribute Types

The most basic attribute types found in video games are:

- *Number*: It includes all type of numbers such as integer, decimals and their respective restrictions. Some of the operations that can be performed on numbers are addition, division, subtraction and logarithm.
- *String*: It includes all types of strings, their formats and their respective restrictions. It has a length attribute which in calculated by the number of characters a string has. Some of the operations that can be performed on strings are append a string to the end of one, return a substring from a string, split the string in other string, etc.
- *Boolean*: It has only true or false as the possible values. All its operations are logic operators of course. There are other types of logic such as first-order logic with a whole exclusive set of logic operators.
- *Enumeration*: It consists of a set of named elements of the enumerator attribute. An enumeration has values that are different from each other, and that can be compared and assigned. For example, the three actions in Rock-Paper-Scissors may be three enumerators named Rock, Paper and Scissors belonging to an enumeration attribute named “player action”.

Concluding this module, to expand on a certain type attribute is the same to develop a new ontology or module to describe this type. The reason for this, as demonstrated, is that attribute types have their own operations and their own particular rules.

7.3 – Event Module

Events are discrete points in the gameplay where the game state changes (BJÖRK & HOLOPAINEN, 2003). Most game engines have an event system, which permits various

engine subsystems to register interest in particular kinds of events and to respond to those events when they occur. A game's event system is usually very similar to the event/messaging system underlying virtually all graphical user interfaces (GREGORY, 2014).

According to GREGORY (2014), an event is comprised of two components: its type (player input, collision, etc.) and its arguments. The arguments provide specifics about the event (Which button was pressed? What objects collided?). Some game engines name events as messages or commands. These names emphasize the idea that informing objects about an event is essentially equivalent to sending a message or command to those objects.

GREGORY (2014) says that most objects in a game do not need to respond to every possible event. Most type of game objects have a relatively small set of events in which they are "interested". This can lead to inefficiencies when multicasting or broadcasting events, because there will be a need to iterate over a group of objects and call each one's event handler, even if the object is not interested in that particular kind of event.

THORN (2013) claims that the relationship between time and objects can be stated as follows: "When X happens, and then does Y". The first part of the statement is referred to an *Event*. The second part of the statement is referred as an *Action*. The event is a notification or moment in time that is raised when an important circumstance arises in the game world (player presses a button to jump). The action is invoked as a response to the event when it occurs to bring about a relevant and specified change in the game world at that time (the character jumps).

Events bring information not available internally to a GO, causing the GO to perform an *Action* according to its state and the information received from the *Event*. The *Action* changes the GO state, possibly triggering new *Events*. Therefore, the purpose of this module is to describe *Events* by detailing their types, information that they bring, the GOs that react to them and the *States* that trigger them. Figure 8 presents a diagram with a simplification of this module which has the following properties:

- An *Event* carries information in the form of *Attributes*, GOs or a combination of both. For example, a touch on a touchscreen generates an Event that informs the coordinates of the touch to the video game. A collision event informs the GOs that collided.

- There are cases that a GO does not respond to an *Event*. For example, in a fighting game the player's character may become stunned and will not respond to any input it would normally react to. This means that there is a set of States in which a GO reacts to an event and one set in which the GO does not react to the event.
- There are two categories of *Events*: internal and external. *External Events* are events caused by actions external to the game such as player inputs. Those inputs reflect the state of the external object that sent those inputs. Therefore, *External Events* are triggered by the *State* of an *External Object*. Those events can occur at any time in the game. It is up to the game logic to handle those types of events. **See Axiom 4**.
- *Internal Events* are events caused by a change of the game state. When a particular GO reaches a determined *State*, it triggers an *Internal Event*. Therefore, *Internal Events* are always triggered from a GO. For example, when a character loses all his life in an action game, it triggers the Game Over event which makes the player lose some progress and forces him to replay the level again. **See Axiom 5**.
- The same *Event* can be handled by more than one instance of a GO. An example is the *Collision* event, which happens when two bounded geometries intersect in a space. The two GOs that own the geometries handle the same *Event*. Another example is the *Double Mario* power-up in *Super Mario 3D World* developed by *Nintendo* where the player controls multiple characters but all of them move according to the same input (if the jump button is pressed, all characters jump at the same time).
- A GO performs an *Action* or sequence of *Actions* as a reaction to an Event. However, the *Action* performed by the GO depends on its *State* and the *Event* information. A common example is fighting games. Different characters have different attacks with different properties that are called by the same button press or sequence. Therefore, multiple *Actions* can be linked to the same *Event* but for each *Action* associated to an *Event*, there are conditions that must be fulfilled. Those conditions can be synthetized in a *State* of a GO.

To conclude this module, a GO changes its own *State* through Actions while it changes the *State* of other GOs by sending *Events* (messages) to other GOs.

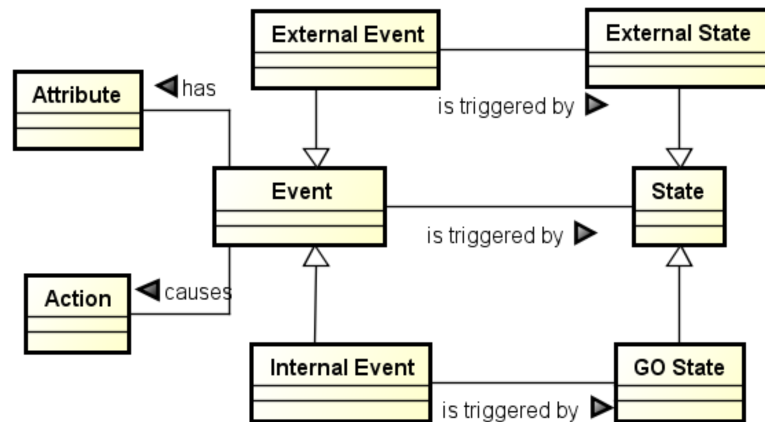

Figure 8 – Event Module

7.4 – Action Module

According to BJÖRK & HOLOPAINEN (2003), players can only change the game state by performing actions. *Actions* available to the player typically change according to the current game state and mode of play. In this ontology, GOs have *Actions* that are called by *Events*. *Internal Events* are triggered by the changes made by *Actions*; this creates a chain reaction where *Internal Events* call *Actions* which triggers new *Internal Events*. Furthermore, the scope of what an *Action* changes can vary, some change only a single *Attribute*, others change several *Attributes*. This means that an *Action* can be divided in smaller ones. It can help developers understand better how an *Action* operates.

Therefore, the first purpose of this module is to describe the conditions necessary for *Actions*. The second purpose is to describe which *Attributes* the *Action* may change. Third, the module will help identify what *Events* an *Action* may trigger by specifying the possible new *States* that may be outcomes of the *Action*. Finally, this module allows developers to make their actions abstract if they are not sure of its inner working but sure of the end result. Figure 9 presents a diagram with a simplification of this module which has the following properties:

- An *Action* may change one or multiple *Attributes*. For example, in the *Pokémon* series there are attacks that besides causing damage also cause status effects. The *Poison Sting* attack has a certain chance of poisoning the opponent. Here, the attack changes the health points (HP) but may change the status attribute.
- Therefore, an *Action* changes the *Attribute* of the performer GO, which can result in a number of possible new *States* that may trigger new *Internal Events* or new *Actions*

from the performer GO. For example, the jump action in *Super Mario Bros.* changes *Mario* state from idle to jumping. It can also change the *Output* of the game (change the image being displayed on the TV screen). The set of new *States* can be called the possible outcomes of the *Action*.

- An *Action* will only happen in a certain *State* of the game. For example, a move action will only be performed if the character is not colliding with a wall which makes impossible to change its location. This *State* works like a condition or constraint.
- An *Action* can have *Parameters* which are an extension of *Attribute*. While an *Event* holds information that the GO processes to determine the *Action* has to take, *Parameters* of an *Action* determine how much and what it changes. An *Action* can have as *Parameters* the *Attributes* of the *Event* that called it. For example, the movement action needs the direction of the analog stick to determine the direction that the character moves.
- An *Action* can have possible previous and next actions if the *Previous Action* outcome *State* is the same as the *Next Action* condition *State*. This way the developer can see the chain reaction that an *Action* may cause. For example, in *Sonic the Hedgehog* the move action is composed of two sequential actions: accelerate and update position. First, the accelerate action updates the speed attribute of *Sonic*. Finally, with the speed attribute updated, *Sonic*'s position attribute is updated. **See Axioms 6 and 7**.
- It is impossible that the condition State is the same as the outcome State. This property exists to avoid modeling endless actions (infinite cycle). **See Axiom 8**.

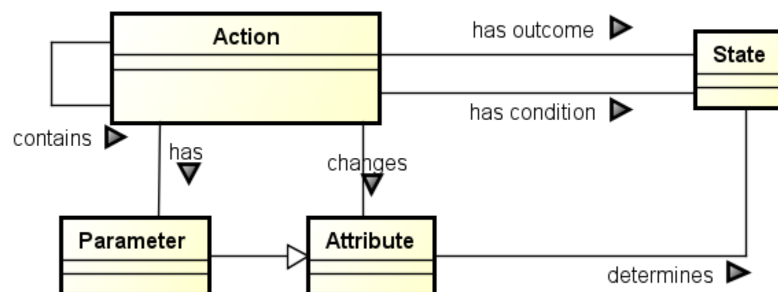

Figure 9 – Action Module

To conclude this module, *Actions* can be seen as a process that an object undergoes in order to handle an *Event* it receives. This is similar to CHANDRASEKARAN *et al.*'s (1999)

claim that “There are processes in which objects participate and that occur over time”. Those processes occur because of events and from those processes new events appear, supporting another claim that “Events can cause other events or states as effects”.

7.5 – State Module

According to BJÖRK & HOLOPAINEN (2003), games are typically structured in different sections, phases or turns where the interface, available actions and information for the player change dramatically. The authors call them different modes of play, which can be seen as constructs to define boundaries between activities within the larger activity of playing a particular game. Typical examples of switching modes of play are the transition from a map view to an inventory screen in a computer role-playing game or turn taking in Chess. States can be used to define how many modes of play a game has by abstracting or adding details. For example, chess can be said to have two modes or states (the player turn or the opponent turn) (BJÖRK & HOLOPAINEN, 2003).

According to SCHELL (2014), it is often useful to construct a state diagram for each attribute to make sure the developer understands which states are connected to which and what triggers state changes. In terms of game programming, implementing the state of an attribute as a “state machine” can be a very useful way to keep all this complexity tidy and easy to debug. GREGORY (2014) says that the state of a game object can be defined as the values of all its attributes.

To demonstrate how states can abstract a rather complex set of interactions, I present Figure 10 which is a flowchart detailing the *Great Sword* weapon move set in *Monster Hunter 4: Ultimate* developed by *CAPCOM*. It was devised by Hiro Rõjin[12] with the intent of helping other *Monster Hunter* players better understand how the weapon works. Therefore, it may have inaccuracies.

In the flowchart, the states represent action done by pressing an input and all transitions represent a player input. It can be a single button press or a combination of them. For example, the player can press X or A after the *Aerial Overhead* to perform the *Super Swipe* or press X and A at the same time after the *Side Slash* to perform the *Rising Slash*.

[12] http://www.capcom-unity.com/monster_hunter/go/thread/view/146585/30422233/updated-move-charts

Figure 10 – *Monster Hunter 4 Ultimate*, *Great Sword* Move Chart

The flowchart leaves many details such as if no input is pressed after an action the character goes to the Idle state automatically or that some character actions take some time to perform. Of course, this is because the intent of the flowchart is to present the move set and that the target audience is aware of some details that can be left out. For example, when performing the *Side Slash* there will be a frame of time in which the character does not accept any input, after this frame of time there will be one in which it accepts player input and if there is no input from the player, the character will go to *Idle* state. As it can be seen, the *Side Slash* can be further divided in more states.

This brings us to an interesting part of the flowchart which is the states inside bigger ones. There are three bigger states that contain smaller states: *Ready to Dodge*, *Pick a Move* and *Overheads*.

- *Ready to Dodge* and *Overheads*: If the player presses B after any character action inside this state, the character will perform a *Front Roll*. This state can also be called a group of states since they share the same resulting states from the same player input. The same applies to *Overheads*, if the player press A after any character action inside this state, the character will perform a *Side Slash*.

- *Pick a Move*: The states *Idle* and *Sword Slap* respond to a number of inputs identically, meaning that for an identical input done in a state, the next state will be identical to both. For example, if the player presses A in either of the two states, the character will perform a *Side Slash*.

This flowchart shows how states can be extremely useful tools for modelling parts of games with simplicity. Thus, this ontology supports the use of states in the description of a video game or parts of it. Therefore, the purpose of this module is to describe *States* and their transitions, which *States* are part of a bigger one, how *States* can change over time and the attribute values that define a *State*. Figure 11 presents a diagram with a simplification of this module which has the following properties:

- A *State* represents a GO state in a frame of time. In the flowchart, the *Side Slash* state represents the frame of time that the character is performing the move; the frame of time is the duration of the animation.

- *States* represent range of values of an *Attribute*. *States* can be created without *Attributes* values, as sometimes it might be more simple just to create a State that represent a complex combination of *Attribute* values or if the developer has no idea what values a particular *State* may have. In the flowchart, the *Idle* state probably has an attribute value that makes the character be in that state. However, it is not the job of the designer but that of the programmer to implement the attributes and the values. That is why it is fine for a *State* to not have attribute values defined in the beginning.

- A *State* can be part of many *States*. Therefore, a *State* can be divided in smaller *States* if it is determined by many *Attributes* or the determining *Attribute* allows range of values. **See Axioms 9 and 10**.

- If the *State* that an *Attribute* determines is part of another *State* then the *Attribute* also determines it. **See Axiom 11**.

- A *GO State* is a *State* of a GO instance. An *External State* is a *State* of an *External Object* instance. **See Axioms 12 and 13**.

- A *State* can have a set of *Next States*. This set is composed of the outcome *States* of *Actions* that have the *State* as a condition *State*. **See Axiom 14**.

- A *State* can have a set of *Previous States*. This set is composed of the condition *States* of *Actions* that have the *State* as an outcome *State*. **See Axiom 15**.
- A *State* can be equivalent to another *State*. For example, the *State* “Health < 4” is equivalent to the *State* “Low Health”.
- *State* extensions can have properties. For example, *States* determined by *Number* attributes can have a lot of properties such as “less than”, “more than”, “equal to”, etc. With this, the developer can define with precision more complex *States*.
- A State should be composed of at least 2 States. **See Axiom 15a**.

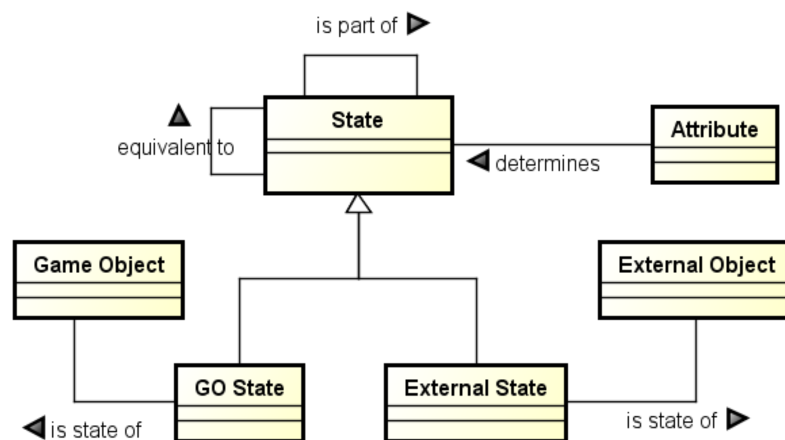

Figure 11 – State Module

To conclude this module, *States* are tools that allow developers to model and analyze modules of the game with ease. The abstraction is really useful to designers and other team members as that allows them to focus on only the relevant interactions of portions of gameplay. Also, *States* are useful to model non-playable characters (NPC) with AI.

7.6 – Space Module

According to SCHELL (2014), every game takes place in some kind of space. This space is the “magic circle” of gameplay. It defines the various places that can exist in a game and how those places are related to one other. He states that we need to strip away all visuals, all aesthetics, and simply look at the abstract construction of a game’s space. Generally, game spaces are either discrete or continuous; have some number of dimensions and have bounded areas that may or may not be connected.

Also, GREGORY (2014) states that most video games take place in a two- or three-dimensional virtual game world that is comprised of numerous discrete elements. When a game takes place in a very large virtual world, it is typically divided into discrete playable regions, which we will call world chunks. Chunks are also known as levels, maps, stages or areas. World chunks are also a convenient mechanism for controlling the overall flow of the game.

This module extends the *Game Object* (GO) Module, so *Spaces* can also respond to events and perform actions. Therefore, the purpose of this module is to describe the different kinds of space that might exist in the game, how those spaces are shaped and how they are connected to each other. Figure 12 presents a diagram with a simplification of this module hierarchy and Figure 13 presents a diagram with a simplification of this module relations. This module has the following properties:

- Number of dimensions. A S*pace* can have from none to three dimensions. Therefore, there are four types: 0D, 1D, 2D and 3D *Spaces*.
- A *Space* must contain a GO at some point in time. There is no point in having a game space if it is not populated by some kind of object. There is no gameplay in an empty space.
- A *Space* can be connected to other *Spaces* through *Connections*. *Connections* are GO extensions. The squares of a chess board can have eight connections for example. Another example are the stages of *Sonic the Hedgehog* developed by *Sega*, each stage is connected to the previous and next stage, each stage containing its own personalized space (in this game case a 2D continuous space). Any kind of *Space* can be connected to another meaning a *0D Space* can be connected to a *2D Space*. As showcased above, a stage is a *0D Discrete Space* that is connected to a *2D Continuous Space*. An example is a stage select menu, the options are the *0D Discrete Spaces* that when selected lead to the actual playable stage.
- A *Space* can have nested *Spaces* (spaces within spaces). A nested *Space* is different from a connected *Space*. A nested *Space* is part of bigger *Space* as it shares the same dimension and operates under the same rules of the bigger *Space*. For example, any

character of a 2D platform is in most cases are simply a 2D collision box inside a greater 2D *Space*.

- A *Space* can have a boundary. A boundary determines the size and shape of a *Space*. A GO is in a *Space* once it is within the *Space* boundaries. Therefore, there are two types of *Spaces*: bounded and unbounded. For each dimension, the composition of boundaries is different such as *0D Spaces* are bounded by their connections with other *Spaces* and *2D spaces* are bounded by a set of interconnected points. An example of *Unbounded Space* would be simple an infinite *2D Space* with the origin coordinate set (x and y are zero). An example of *Bounded Space* would be a geometrical object like a triangle or cube.
- Unbounded spaces must contain Bounded spaces. Gameplay can happen in an infinite space but human players can only control and recognize entities with limited shapes.

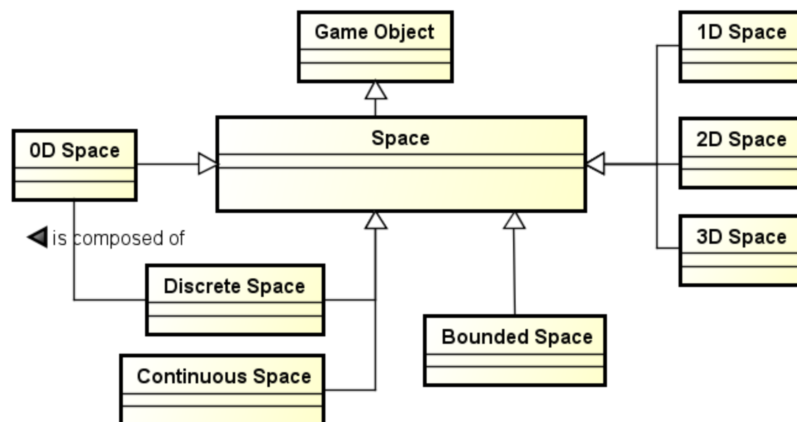

Figure 12 – Space Module Hierarchy

This element has the following extensions: *Discrete Space* and *Continuous Space*. The main difference is that the position of a GO in a *Discrete Space* must point to a *0D Space*, in other words it is absolute. While in a *Continuous Space*, the position attribute simply points to a coordinate in the space and in that space there can be an infinite number of points. In other words, the number of possible values of the position attribute is limited in a *Discrete Space* while in a *Continuous Space* is infinite.

7.6.1 – Discrete Space

Discrete Spaces are a set of simple *0D Spaces* that can be organized as a dimensional space and, obviously, can be composed of other *Discrete Space*. *0D* Spaces must be

connected to each other to be recognized as a *Space*. I will refer to the individual *0D Spaces* as *Nodes* because this type of space can be seen as a graph. A *Discrete Space* can have none to many dimensions depending on how their nodes are organized. **See Axioms 16 and 17**.

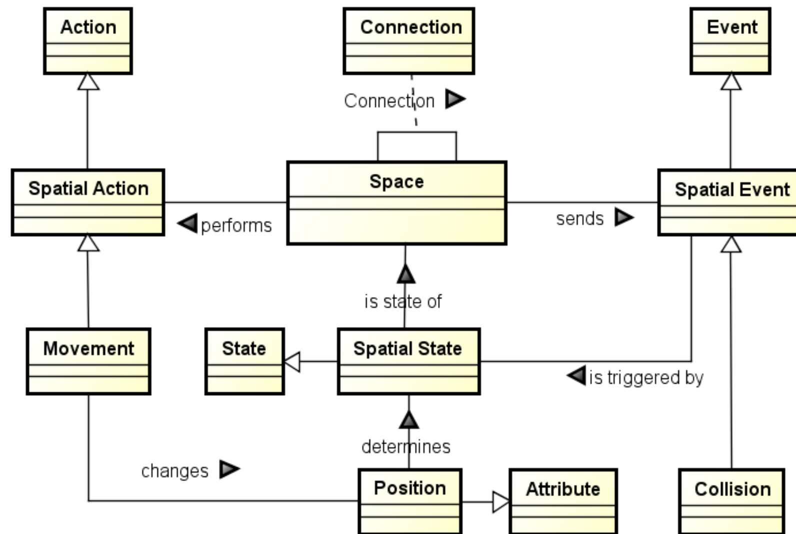

Figure 13 – Space Module Relations

0D Discrete Spaces are organized in a way that does not resemble a dimensional space or are hard to classify as a dimensional space. Most common examples are trees and graph. Menu can be organized as a tree of options. These types of space can have boundaries. For example, in a tree the developer can determine the minimum and maximum value of the width and depth.

1D Discrete Spaces resemble a line or an array of objects. There are games such as Monopoly where the board can be treated as a line. The maximum number of connections is two, one for the previous *Node* and one for the next *Node*. A boundary can be easily set with a start *Node* and a finish *Node*.

2D Discrete Spaces resemble a *Continuous 2D Space* or a matrix of objects. Most common examples are board games such as chess and turn-based strategy RPGs such as *Final Fantasy Tactics*. Boundaries can come be set up in two methods: there is set of *Nodes* with specific properties that are different from the other sets or the *Connections* of certain *Nodes* have special properties.

An example for the first method can be found in *Advance Wars* and for the second method can be found in *Fire Emblem Awakening*, both developed by *Intelligent Systems*. Both

games are turn-based board games with square tiles. In *Advance Wars* each square represents a different kind of terrain such as sea and grasslands and there are units that can only traverse some type of squares. Just by making a stage with a set of grassland squares surrounded by sea squares is a way of setting a boundary since infantry units cannot traverse sea squares, blocking them. In *Fire Emblem Awakening*, there can be squares separated by a wall but the wall does not occupy a square. Certain units cannot move to the other square or attack units located in it because of the wall. The reason for this is that the *Connection* between the two squares has a special property or is of a different type.

The same reasoning of *2D Discrete Spaces* can be applied to *3D Discrete Spaces*, the only difference between them is that the number of *Connections* rises significantly and what it is trying to resemble. There are other *Discrete Spaces* that are hard to classify such as hexagonal boards which are composed of hexagons and can have a maximum number of six connections.

In *Discrete Spaces*, the movement action result can be influenced by attributes such as speed and acceleration or number of spaces an object can move. Examples include the constant movement of the player character in *Pokémon Red*, accelerated movement in *Tetris*, as the time passes the velocity in which the blocks fall gradually increases and pre-determined range of movement of chess pieces. Collision detection is influenced by the *Connection* of the *Discrete Spaces* or the shape of the *Bounded Space*, a collision is detected if part of one space is inside of another or if the *Connection* prevents the movement. Examples are board games such as chess (only one piece can occupy a square and if a piece) and strategy games such as *Final Fantasy Tactics* where if many characters are in the trajectory of a gun attack (it is a straight line), only the closest will be hit.

7.6.2 – Continuous Space

Continuous Spaces must have a number of dimensions different from zero and it is composed by an infinite number of points. Those types of space are used to enable real-time gameplay and enable more sophisticated mechanics such as real-time physics and collisions to enable games that closely resemble the real world (simulations). GOs that are within a *Continuous Space* have a position attribute and are represented in that *Space* with a *Bounded Continuous Space*. A point is the smallest *Bounded Continuous Space* possible.

A *1D Continuous Space* is only used for gameplay mechanics. Most prominent examples are *Pong* and *Breakout* where the player moves a paddle in just one axis (in *Pong* the paddle moves up or down, in *Breakout* the paddle moves left or right). *1D Bounded Spaces* are straight lines with the direction in the X axis or Y axis. Of course, there are curves or inclined straight lines but the points they occupy are vary in two dimensions. It would be a different case, if a GO would move within the curved line, as it would only move left or right along the line trajectory. This would be a case of 1D gameplay.

2D and *3D Continuous Spaces* are used to, besides designing gameplay mechanics, display the game space for the player to be able to understand what is happening in the game. There are games that have 2D gameplay mechanics and are presented through *2D Assets* (*Super Mario World*); have 2D gameplay mechanics and are presented through a *3D Assets* (*Trine* and *Klonoa*), some of these games are called 2.5D games because of this mix; have 3D gameplay mechanics and are presented through a *3D Assets* (*Super Mario 64*); have 3D gameplay mechanics and are presented through a *2D Assets* (*Streets of Rage* and *Castle Crashers*), most classic beat'em ups fit in this category as characters can move in freely in a space and jump.

2D and *3D Bounded Continuous Spaces* are geometric shapes. Many two-dimensional geometric shapes can be defined by a set of points or vertices and lines connecting the points in a closed chain, as well as the resulting interior points. Such shapes are called polygons and include triangles, squares, and pentagons. Other shapes may be bounded by curves such as the circle or the ellipse. Many three-dimensional geometric shapes can be defined by a set of vertices, lines connecting the vertices, and two-dimensional faces enclosed by those lines, as well as the resulting interior points. Such shapes are called polyhedrons and include cubes as well as pyramids such as tetrahedrons. Other three-dimensional shapes may be bounded by curved surfaces, such as the ellipsoid and the sphere.

In *Continuous Spaces*, the movement action result is influenced by attributes such as speed and acceleration. Examples include the constant movement of the player character in The Legend of Zelda and accelerated movement in racing games. Collision detection is influenced by the shape of the two *Bounded Spaces*, a collision is detected if part of one space is inside of another. Examples include fighting games which feature different hitboxes for each character attack.

7.6.3 – Spatial Attributes, Actions, States and Events

There are four extensions derived of the creation of the *Space* module:

- Spatial Attribute: Those are *Attributes* that define the *Space* properties. Those properties include position, size, speed, acceleration, etc. **See Axiom 17a**.
- Spatial States: *States* determined by *Spatial Attributes*. In most cases, *Spatial States* are determined by comparing two Spaces positions and their shapes overlap. Those *States* can be a collision, distance between two *Spaces*, the connection between *Discrete Spaces* and the relative position of an *External Object*. **See Axiom 18**.
- Spatial Action: *Actions* that modify *Spatial Attributes*. An example is to change the position (movement) or shape (size or border) of the *Space*. How they work depends on the dimension of the space and if it is discrete or continuous. **See Axiom 19**.
- Spatial Event: *Events* that are triggered by *Spatial States*. In most cases, those *Events* are collisions between two *Bounded Spaces* or the position of *External Objects* (position of an analog stick). **See Axiom 20**.

7.7 – Time Module

Video games are real-time, dynamic, interactive computer simulations. As such, time plays an incredibly important role in any electronic game. One of the changes in the states of objects that occur over time are animations; without a conception of time, no animation would be possible (THORN, 2013). According to GREGORY (2014), there are many different kinds of time to deal with in a game engine: real time, game time, the local timeline of an animation, the actual CPU cycles spent within a particular function, and the list goes on.

According to SCHELL (2014), video games also give players the chance to do something that cannot be done in the real world: control time. This can happen in a number of ways: time can be stopped completely, as when a “time-out” is called in a sporting match or when the “pause” button is pushed on a video game. Time can be accelerated in games like *Brave Fencer Musashi* developed by *Squaresoft* where the player can fast-forward time by making the main character sleep. Time can be rewinded in many ways on video games. For example, every time the player dies in a video game and return to a previous checkpoint is like going back to a point in time. *Pushmo* developed by *Intelligent Systems* features a

mechanic where the player can rewind a certain amount of time to undo a mistake like a wrong jump. In this video game, the player can rewind time like a video but with a limit.

This module's root element is *Time* and it extends the *Attribute* Module, so *Time* can be manipulated by actions and trigger events. Therefore, the purpose of this module is to describe the different kinds of time that might exist in the game, how time can be measured, how it can be manipulated and how it affects other GOs. Figure 14 presents a diagram with a simplification of this module hierarchy and Figure 15 presents a diagram with a simplification of this module relations. This module has the following properties:

- *Time* is an extension of a *Number* attribute. A game can have its time measured by seconds or number of days. Also, it can be changed using the same type of operations. For example, in *Time Crisis* when the player beats all enemies the time is extended.
- A *Time* attribute can be equivalent to a certain amount of other *Time* attribute. For example, one minute is equivalent to sixty seconds.
- *Actions* that take a certain amount of *Time* to conclude are called *Timed Actions*. For example, the *Side Slash* action from the character of *Monster Hunter* takes some time to conclude (in this case, the duration of the *Action* is equal to the duration of the animation).
- A *Timed State* is a *State* determined by a *Time* attribute. It simply means that the *State* has a fixed duration and it will change without input from the player. For example, in *Super Mario World* when *Mario* touches a star, he becomes invincible. This state has certain duration and when the time elapsed is equal of the duration *Mario* reverts to the vulnerable state. **See Axiom 21**.
- A *Timed Action* has a *Timed State* since it has a *Time* attribute to measure its duration. **See Axiom 22**.
- *Events* that are triggered by *Timed States* are *Timed Events*. For example, in *Sonic the Hedgehog*, there is a time limit for Sonic to remain underwater. If the time is below 5 seconds the background music changes to a menacing tune and a countdown appears indicating the time left. If it reaches zero, the player loses a life. **See Axiom 23**.

- According to GREGORY (2014), because video games are dynamic, time-based simulations, a *GO State* describes its configuration at one specific instant of time. Thus, a group of *States* can represent a frame of time. For example, in the *Legend of Zelda: Wind Waker* a day is divided in two phases: daytime and night. When there is a *State* that represents the daytime portion of the game, it represents the frame of time between start of daytime phase and end of daytime phase which have timestamps associated to them.
- A *Timed State* can be divided further in time slices. With this, the developer can model events that happen through the duration of an *Action* for example.

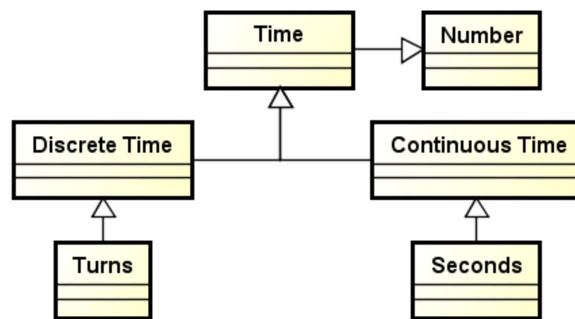

Figure 14 – Time Module Hierarchy

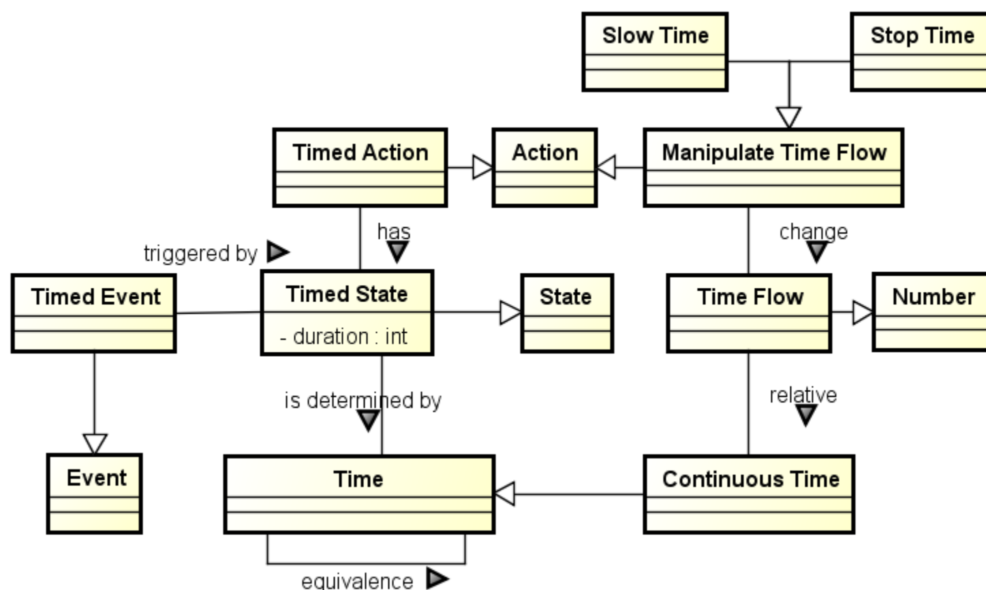

Figure 15 – Time Module Relations

In the following sections I will discuss the following pair of extensions: *Discrete Time* and *Continuous Time*. Also, I will talk a bit more about time manipulation.

7.7.1 – Discrete and Continuous Time

SCHELL (2014) asserts that time can also be discrete or continuous like space. The unit of discrete time in a game: is called "turn". Generally, in turn-based games, time matters little. Each turn counts as a discrete unit of time, and the time between turns, as far as the game is concerned, does not exist. Chess games, for example, are generally recorded as a series of moves, with no record of the amount of time that each move took, because real clock time is irrelevant to game mechanics. Schell adds that there are many games that are not turn based, but instead operate in continuous time. Most action video games are this way, as are most sports. And some games use a mix of systems. Tournament chess is turn-based but has a continuous clock to place time limits on each player.

There are *Discrete* and *Continuous Time* units which are similar with the exception of the equivalence relationship between them. When there is equivalence, for example, of one turn being equal to forty seconds, it means that the duration of a turn in the game is of forty seconds. It is the same by saying that a turn of tournament chess has duration of forty seconds.

The main difference between *Discrete* and *Continuous Time* is that the former advances in accordance to the game rules while the latter advances in real-time fashion, in other words it is updated as fast as the computer processor allows. Therefore, every GO that operates in *Continuous Time* has a *Time Flow* which is a *Number* attribute. It is default value is one which means that the *Time* attribute advances just like the real-world time (one second in the game is one second in the real-world).

Therefore, manipulation of *Continuous Time* is different from *Discrete Time* because of the *Time Flow*. In real time video games, actions such as movement operate under a certain speed, so if the *Time Flow* is doubled the speed is doubled (two seconds in the game is one second in the real-world). However, there are cases where the developer only needs to slow or accelerated one GO (two seconds for the GO is one second in the game). *Dead Space* developed by *Visceral Games* provides a great example of slow and accelerated GOs. Once the player reaches a point in the game he obtains the *Stasis Module*[13]. It is a device capable of producing a temporary time dilation, making objects move at an extremely slow rate for a period of time. With this power, the player can make enemies move in slow motion. Later in

[13] http://deadspace.wikia.com/wiki/Stasis_Module

the game, the player encounter enemies called *Twitchers*[14] which are accelerated enemies as they move at high speeds compared to the rest of them. The use of the *Stasis Module* is necessary to bring the *Twitchers* to normal speed.

The *Time Flow* is related to a *Continuous Time* attribute, so *Timed Action* and *Timed Events* that use the related time for their operations will have their duration changed. Also, every action that manipulates *Video* and *Audio Output* operates in *Continuous Time* because those types of outputs are sent at real time to the player.

7.7.2 – Rewinding Time

Rewinding the time of a video game can be a complex process depending of the kind of the game. To effectively rewind time in video games, the previous states of the game must be stored and accessed in order. It makes sense since a *State* represents an instant of time (by having stored the timestamp of the *State*) and the *Next State* is the next instant of time. By chaining together the states, a timeline of the game can be constructed where the player can explore and return to the point he desires. Action games can feature time rewind mechanics but they have a limit of how much they can rewind because of their complex *States*. Many games implement time rewind by recording the player actions and GO actions instead of the game states because in the majority of cases the previous states can be simply reached by undoing or reversing actions.

[14] http://deadspace.wikia.com/wiki/Twitchers

Chapter 8 – Conceptualization of External Modules

The objective of this chapter is to organize and structure the knowledge acquired during the specification. In this chapter, the external modules of the Video Game Development Ontology will be conceptualized. The conceptualization process is the same as in Chapter 7.

8.1 – External Object Module

This is the root module of the external part of the ontology. It is from this module that other external modules will be born. Some of the *Game Objects* (GO) must communicate with objects from the real world to deliver the intended experience to the player. The designers need to know the *External Objects* (EO) that are part of the video game experience; those objects include the hardware that the game runs on, the software that the video game communicates with and the human players that interact with the game. With this module, all the components that are not part of the game world are described and their interactions with the game world are detailed.

Therefore, the purpose of this module is to describe the EOs by detailing the relations between different objects, the states that an external object has, the inputs it sends to GOs and the outputs it receives from GOs. This module is solely composed of the EO element and has relations with all external modules. Figure 16 presents a diagram with a simplification of this module hierarchy and Figure 17 presents a diagram with a simplification of this module relations. This module has the following properties:

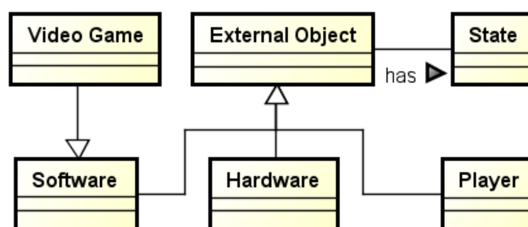

Figure 16 – External Object Module Hierarchy

- An EO has *States*. The designer does not need to know the intricacies of an EO but only the potential *States* of the EO that might affect the game world. This also implies that an EO can have *Attributes*. While the game cannot have access to the EO, it is useful to assign *Attributes* to relevant information about the EO for the gameplay.

- An EO sends *Input* to a GO and receives *Output* from the GO. In turn, the GO receives the *Input* from the EO and sends *Output* to the EO.

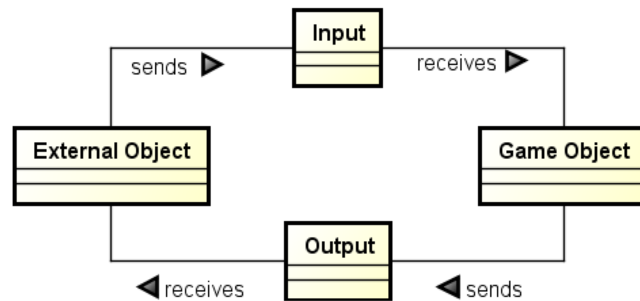

Figure 17 – External Object Module Relations

8.2 – Hardware Module

Video games are software programs that run on electronic hardware such as computers, tablets, consoles, handhelds, smartphones, etc. Also some of them need specialized hardware to enable the interaction of players with the game and vice versa. There are many types of such hardware:

- Input: controller, keyboards, touchscreens, cameras, etc.
- Output: TVs, headphones, LCD displays, surround sound systems, etc.
- Input and Output: act as both input and output such as Internet connections (cable or wireless) used for multiplayer games where data travels back and forth between players.

Also, there is hardware which features a lot of hardware components in a single package. Handheld consoles are such example. The *Nintendo 3DS*, for example, features an analog disc, ten digital buttons, one directional pad, two screens with one being a touchscreen, two cameras, microphone, speakers, Wi-Fi connection and much more.

This module's root element is *Hardware*. *Hardware* are physical objects while *Software* are abstract objects (not located in the real world) such as video game and operational systems. Therefore, the purpose of this module is to describe the different kinds of hardware that are necessary to play certain video games, the software that runs on the hardware, the physical inputs a hardware can send to the video game and the physical outputs

a hardware can receive from the video game. Figure 18 presents a diagram with a simplification of this module which has the following properties:

- A *Hardware* can be composed of other *Hardware* or be part of one. The *Nintendo 3DS* is an example of that.
- A *Hardware* can run many *Software*. A *Video Game* is a *Software* obviously. *Software* is an important element since some *Output* from the *Video Game* cannot be handled directly by the *Hardware*.
- A *Hardware* only sends *Physical Input* to a GO. A button press from a controller makes a character perform an action (jump, punch, kick, etc.). **See Axiom 24**.
- A *Hardware* only receives *Physical Output* of a GO. The image players see on the display screen is provided by the *Video Game* software. **See Axiom 25**.

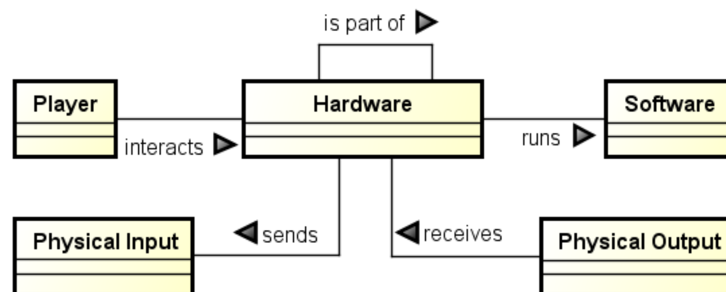

Figure 18 – Hardware Module

8.3 – Software Module

In many cases, a *Video Game* must communicate with other *Software* that runs on the same *Hardware*. Common examples are operating systems (*Windows*), client applications (*Steam*), servers (multiplayer games), etc.

This module's root element is *Software*. Therefore, the purpose of this module is to describe the different kinds of software that communicates with the videogame, the hardware it runs on, the non-physical inputs a software can send to the video game and the non-physical outputs a hardware can receive from the video game. Figure 19 presents a diagram with a simplification of this module which has the following properties:

- A *Software* only sends *Non-physical Input* to a GO. For example, in an online multiplayer game the actions of other players are *Non-physical Inputs*. **See Axiom 26**.

- A *Software* only receives *Non-physical Output* of a GO. For example, in an online multiplayer game every actions the player makes is sent as a *Non-physical Output* to the server. **See Axiom 27**.
- A *Software* can be divided in smaller modules. Therefore, a *Software* can be composed of other *Software* that are its modules or be a module of another *Software*.

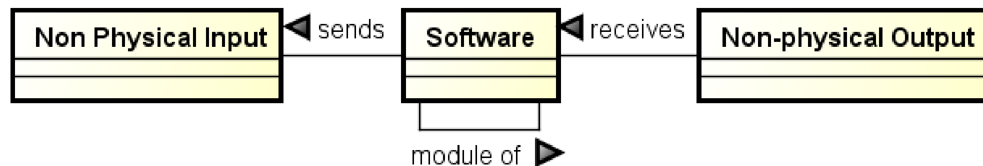

Figure 19 – Software Module

8.4 – Player Module

Human players are the necessary component to bring the game to life. They are the origin of the inputs that a video game receives. It is important to describe the player in the ontology as it is important to know which GOs the player can control, the inputs available and the output he receives. The importance doubles when describing multiplayer games as the GOs will have different players controlling them and each player will have its own output.

This module's root element is *Player*. Therefore, the purpose of this module is to describe the inputs that the player make, the GOs that a player controls, the actions that the player can do with the controlled GO and the output that the player receives (what he feels, sees and hears). Figure 20 presents a diagram with a simplification of this module which has the following properties:

- A *Player* sends *Input* and receives *Output* through interaction with a *Hardware*. **See Axiom 28**.
- A *Player* sends *Player Input*. *Players* can send *Inputs* through interaction with different *Hardware*. *Player Inputs* are associated to a *Player* and they change a *Player Object* which can be only controlled by one *Player* at an instant of time (the same *Player Object* can be controlled by another *Player* or AI). **See Axiom 29**.
- GOs that handle *Player Inputs* are *Player Objects*. Thus, the *Player* controls *Player Objects* and the *Actions* of those GOs that are caused by *Player Inputs* are *Player Actions*. **See Axioms 30 and 31**.

- A *Player* receives *Player Output*. *Player Output* includes camera views (a game can have different types of camera, allowing different views of the game world), views for different players (split-screen in *Mario Kart* is an example of a single output that is divided in two parts, one for each player) or a single view for both players (fighting games such as *Street Fighter*). This *Output* is transmitted from monitors or speakers, for example. **See Axiom 32**.
- A *Display Space* (will be detailed in Section 8.7) that sends *Player Output* is a *Player View*. In short, they represent what the player sees in an instant of the game. Examples are menu screens, cameras, top-down view from some RPGs, side view from the side-scrollers, mini-maps from the HUDs (heads-up displays). **See Axiom 33**.

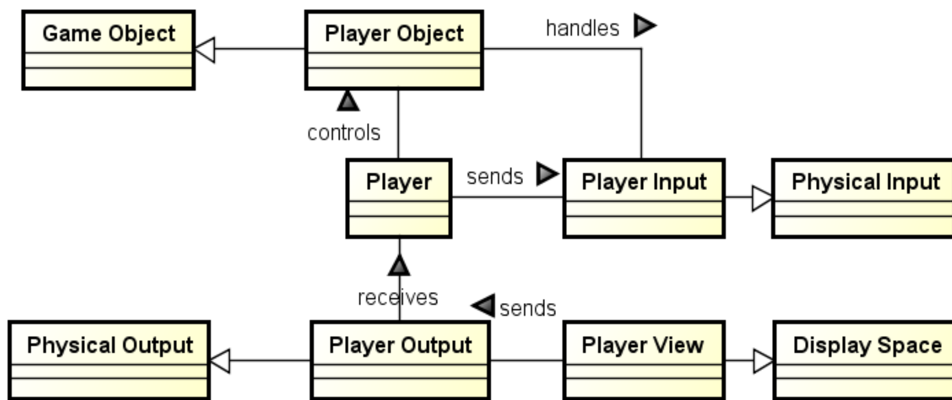

Figure 20 – Player Module

8.5 – Input Module

Players change the game state through inputs performed in hardware such as controllers, keyboards, mouse, etc. Besides button presses, inputs can come in many forms such as cameras which provides images for the game, microphones which provide sound for the game, touchscreens where the player can interact with objects present in such screen and many more. It is the job of the game logic to process the data provided by those inputs and inform the appropriate GOs to perform the appropriate actions considering the input information and the game state.

This module's root element is *Input* and it is an extension of *External Event*, so GOs react to those *Inputs* and perform the appropriate *Actions*. Therefore, the purpose of this module is to describe the different kinds of inputs, the values they can take and the inputs that

a piece of hardware contains. Figure 21 presents a diagram with a simplification of this module which has the following properties:

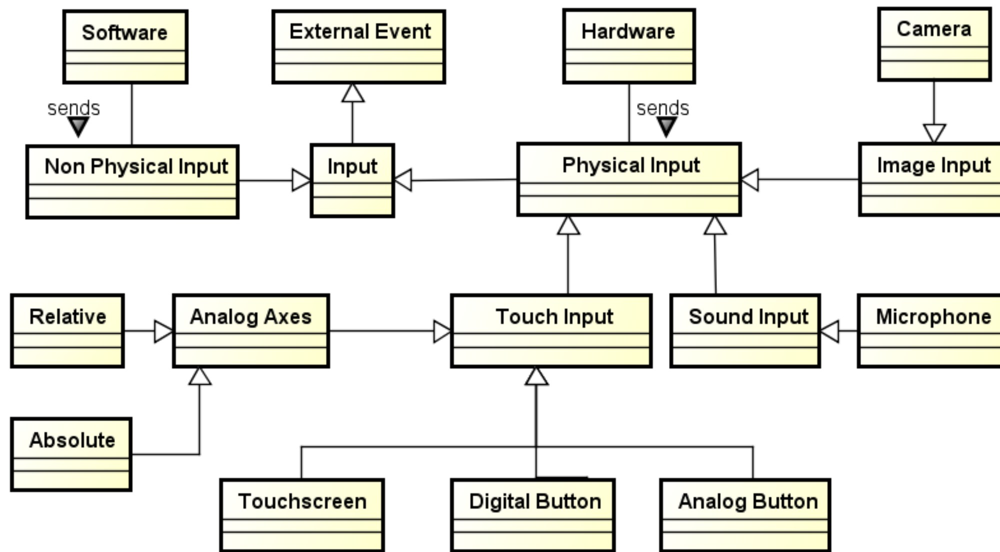

Figure 21 – Input Module

- As an *Input* is an extension of *Event*, it contains *Attributes*. For example, GREGORY (2014) states that digital buttons can only be in one of two states: pressed and not pressed (zero or one values). Analog inputs can take on range of values rather than the two values of a digital input.

- An *Input* can be composed of other *Inputs*. This means that it is triggered by multiple *External States* at the same time. It is especially useful to detect special inputs such as chords (multiple buttons pressed together).

- Special inputs such as *sequences* (button pressed in sequence within a certain time limit) and *gestures* (sequence of inputs from the buttons, sticks, accelerometers, etc.) are dealt within the game logic as a sequence of *Events* (*sequences* are a sequence of *Timed Events*).

In the following sections I will discuss the following two categories of *Inputs*: non-physical and physical.

8.5.1 – Non-physical Input

Non-physical Inputs are provided by the operating system or electronic signals from the hardware that the video game is running on. Therefore, those *Inputs* are sent by *Software* running in the *Hardware*. **See Axiom 34**. Examples of this are:

- The achievements system of the *Steam* platform that notifies the player about obtaining an achievement during gameplay;
- Online multiplayer has other players inputs provided by an Internet connection;
- Usage of the *Nintendo 3DS* system clock in the game logic by *Animal Crossing: New Leaf* developed by *Nintendo* to allow time in the game pass in sync with the time in the real world, so events like the time of opening and closing of shops in the game are tied with time in the real world. Of course, if the player manipulates the *Nintendo 3DS* clock, he can visit in the morning shops that would be only open in the night.
- Some video games allow the players to use music from their computer. An example is *Beat Hazard* developed by *Cold Beam Games* where the levels have their content generated depending on the music the player has chosen. In other words, there can be *File Inputs* provided by the operating system.

8.5.2 – Physical Input

Physical Inputs are provided by the interaction of a human player with an input device such as controllers or keyboards. **See Axiom 35**. There are three categories of *Physical Input*:

- Sound: They are solely comprised by microphones. Example of game that uses one is *Phoenix Wright: Ace Attorney* where the player can make the main character shout objection by shouting at the *Nintendo DS* microphone.
- Image: They are solely comprised by cameras. Example of game that uses it is *Pushmo* which the player can use the Nintendo 3DS camera to scan QR codes in order to obtain other player created levels.
- Touch: The majority of ways used to interact with video games are touch-based. Controllers, keyboards, mouse, touchscreens are examples.

According to GREGORY (2014), most of the physical inputs fall in the following categories: digital and analog. Here, he is referring to touch based input such as buttons and sticks. Digital buttons can only be in one of two states: pressed and not pressed. Analog inputs can take on range of values rather than the two values of a digital input and are often used to represent the degree to which a button is pressed. Types of *Touch Input* are:

- Digital Buttons: click of a mouse, pressing the key of a keyboard, pushing the A button of the *Nintendo 3DS*.
- Analog Buttons: some of the buttons of the *DualShock 2* are analog. An example is *Metal Gear Solid 2* developed by *Konami* on the *PlayStation 2*. It uses pressure-sensitive button data in aim mode to tell the difference between releasing the X button quickly (which fires the weapon) and releasing it slowly (which aborts the shot) (GREGORY, 2014).
- Analog Axes: analog inputs used to represent the two-dimensional position of a joystick (which is represented using to analog inputs, one of the x-axis and one for the y-axis. Thus, they are called *axes* (GREGORY, 2014). Analog axes can be divided in two categories: absolute and relative.
- Absolute Analog Axes: The position of an analog button, trigger, joystick or thumb stick is absolute meaning that there is a clear understanding of where zero lies (GREGORY, 2014). Example is the *Dualshock 2* analog sticks.
- Relative Analog Axes: For theses axes, there is no clear location at which the input value should be zero Instead, a zero input indicates that the position of the input has not changed, while nonzero values represent how much the position has changed. Examples include mice, mouse wheels and track balls (GREGORY, 2014).
- Touchscreen: The input value is the point where the player touched the screen. Examples include smartphones, the *Nintendo DS* and *3DS* lower screens.

8.6 – Output Module

As players change the game state through inputs, they must receive immediate feedback of their actions. When they press a button, they should see their character swinging a sword at an enemy, hear the sound of when the sword hits the enemy and feel the rumble of

the impact made by the sword. The game logic prepares the appropriate output data that reflects what is happening in the game world and sends to the hardware. With the output provided to the player he will be able to interact properly with the video game.

This module's root element is *Output* and it is an extension of the *State* element because part of the state of the game is outputted in an appropriate format for an *External Object*. The *Output Action* changes the *Output* (the *Attributes* that determine it) that is sent to the target *Hardware*.

Therefore, the purpose of this module is to describe the different kinds of outputs, the actions that change those outputs and the hardware which those outputs are sent to. Figure 22 presents a diagram with a simplification of this module which has the following properties:

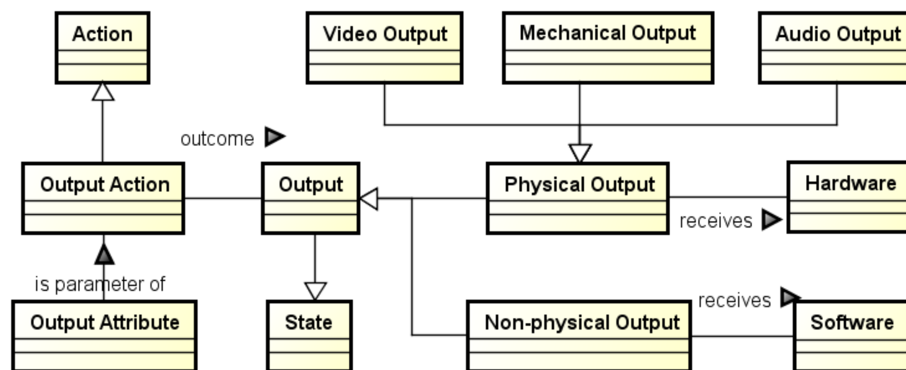

Figure 22 – Output Module

- An *Output* being a *State* means that certain information of a GO will be made available to *External Objects*. For example, *Video Output* concerns with how a GO is displayed to the *Player*. Also, an *Output* is a *State* that has no effect on the game logic, so it has no next *State* and it cannot be a condition of an *Action*. **See Axiom 36**.
- Because an *Output* is a State, it is determined by *Attributes*. However, those have no effect on the game logic. In general, they are matrixes composed of RGB cells (video output) or audio streams dedicated to background music or sound effects. Therefore, there is no need to detail the *Attributes* that determine an *Output*.
- *Actions* that have an *Output* as an outcome are *Output Actions*. Output Actions have no next actions because of the fact that an *Output* has no next *State*. Some examples are drawing actions and playing audio. **See Axiom 37**.

- *Attributes* that are parameters of an *Output Action* are called *Output Attributes*. Simple examples are color and position, because when their values are changed the player will perceive them immediately through the *Output*. It should be noted that *Actions* that change *Output Attributes* **are not** *Output Actions* but they are considered to affect the *Output*. **See Axioms 38 and 39**.
- Because an *Output* is a *State*, it can be composed of other *Outputs*.
- A GO sends an *Output* of it has an *Output Action*. **See Axiom 40**.

8.6.1 – Non-physical Output

Non-physical Outputs are sent to the operating system or the hardware that the video game is running on. They are not discernable for the human player. Therefore, those outputs are sent to the *Software* running in the *Hardware*. **See Axiom 41**. Examples of this are:

- The achievements system of the *Steam* platform is notified by the game when the conditions to unlock an achievement are fulfilled;
- Online multiplayer has the player sending outputs (which become inputs to other players) through an Internet connection;

8.6.2 – Physical Output

Physical Outputs are discernable to the human player. *Physical Output* can only be received by Hardware. **See Axiom 42**. They come on three categories:

- Mechanical Output: players can feel by touch what happens in the game. The most common occurrence is the use of rumble. The *Dualshock* controller of the *Playstation* is notorious for its rumble feature.
- Audio Output: players can hear what happens in the game. Features music, sound effects, voice recordings, etc.
- Video Output: players can see what happens in the game. Features 2D images, 3D models, textures, videos, special effects, etc.

The *Output Actions* for each category of *Output* are different from each other as they change different kinds of outputs. In the next section, I will describe *Audio Output Actions*. *Video Output Actions* will be described in Section 8.7.

8.6.3 – Audio Output Actions

These type actions include playing, slowing, fast-forwarding i.e. manipulation of audio files. Of course, audio manipulation is actually more complex than simply playing some audio files at any time and at the same time as others but it is not on the scope of this ontology to detail it. Examples:

- Change the volume of the audio output or of the audio file (different files operate at different volumes);
- Loop a certain audio file. A lot of games loop background music to maintain a certain feel to the scenario;
- Stop a certain audio file of playing in the output (other sounds can be played).

8.7 – Video Output Module

The *Video Output* is the most important form of conveying to the player what is happening in the game. It can be simple as drawing simples 2D shapes in games like *Tetris* or be an extremely complex task such as rendering large 3D worlds and animate the characters within it in games like *Grand Theft Auto*. Also, what the player sees in the screen must faithfully reflect the game logic that is hidden from the player.

This module's root element is *Video Output* and it is an extension of the *Output* element. The *Video Output Action* is an *Action* has as an outcome the *Video Output* that is sent to the target *Video Hardware*. **See Axiom 43**.

Therefore, the purpose of this module is to describe the different kinds of video outputs, the actions that change those outputs, the hardware which those outputs are sent to and the logic behind the construction of such output. Figure 23 presents a diagram with a simplification of this module which has the following properties:

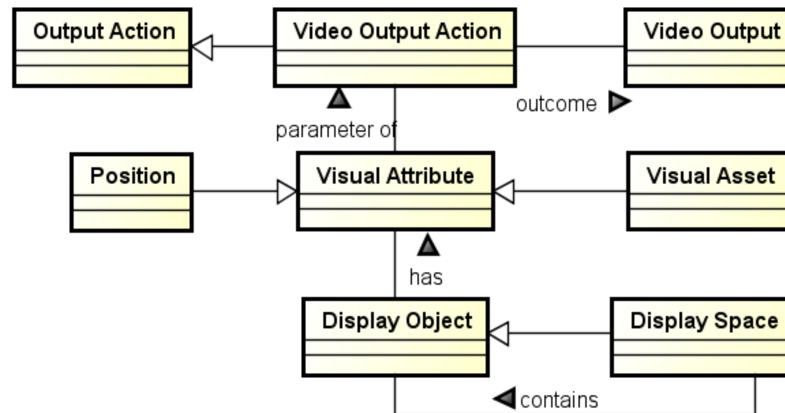

Figure 23 – Video Output Module

- *Display Spaces* are *Spaces* that send *Video Output*. They are what the *Player* sees. They can represent the entire game space (*Tetris* and *Pac-Man* are examples) or part of it (most 3D games and 2D platformers such as *Sonic* are examples). **See Axiom 44**.
- A *Game Object* is a *Display Object* if it sends *Video Output*. **See Axiom 45**.
- *Display Space*s contains *Display Objects*. Such type of *Space* has to be populated with objects that can be seen by the *Player*. There can be objects that are not displayed at all but are part of the gameplay (checkpoints are invisible but are a vital part of the game and occupy the game space).
- *Display Objects* may have *Visual Assets*. Those include 3D models, 2D images, etc.
- *Visual Attributes* are *Attributes* that are parameters of *Video Output Actions*. For example, the size attribute of a *Game Object* will change the rendered 3D Model. **See Axiom 46**.
- *Visual Assets* are associated to one or *more States*. When *Mario* is in a jumping state the sprite presented to the player is of *Mario* jumping, when *Mario* is in a crouched state the sprite presented is of *Mario* crouching.

Space plays a great role in the visual presentation of the game. For example, changing the size or the position of an object can cause it to change how it is presented to the player (a bigger character or the character disappearing from the player sight because of it is behind a wall). However, a *Spatial State* is not a part of a *Video Output*. *Spatial States* will be commonly seen as conditions for *Video Output Actions*.

Actions involving the visual presentation of the game are more complex as there are important characteristics surrounding *Video Output* and its *Actions*.

The first is that the *Video Output* that a *Video Hardware* receives is a 2D image (I will not consider 3D or Virtual Reality displays that have different formats).

The second is that the display screen sometimes cannot display the entire game space. Therefore, *Video Output Actions* can only be performed by GOs within the *Display Space*. *Display Spaces* are determined by cameras, be it 2D (it would be simply a 2D Bounded Space) and 3D. Cameras can be manipulated as they can have their position changed and the image generated by them can be manipulated. In *3D Spaces*, cameras become more complex as the new dimension introduces concepts such as depth, angles, zoom and much more. 3D cameras can be a module of their own as there are different types of cameras such as ones that the player can control at will, ones that move through a predetermined path, ones that remain in a fixed angle and height (real time strategy games such as *Warcraft 3*) and much more.

The third is that the *Video Output* is a 2D Image that can be separated in parts. For example, in many games there is an interface telling how much life the character has or a map indicating where the player is in the game world. Those interfaces generally are put in front of the displayed game world, this is much more apparent in FPS (First Person Shooters) game where the HUD (Heads-up display where important information such as health and ammunition is displayed) never changes angle and is always seen in the same position the entire game. This means that the resulting *Video Output* can be divided in layers (*Display Spaces*) where the image from the camera is drawn first and the image which only contains interface elements is put above the first image (thus being called layers). For each above layer, any pixel is transparent if it has no value determined. 2D games can also have their video output constructed like this as there are interface, foreground, background, characters and many other layers. This helps in separating the GOs in groups that are responsible for a layer of the output. Finally, this means that a *Video Output* can be composed of several *Display Spaces*.

Examples of *Video Output Actions*:

- Render a *Visual* or *Textual Asset*. For text to be rendered, a *Font* is necessary;
- Render a geometric shape. It could be 2D like a triangle or 3D like a pyramid.

8.8 – Asset Module

As it was stated in Section 3.2.1, assets are elements that are used in the output of video games. They differ from normal file attributes as they are associated to the GOs that compose the logic of the game. Finally, assets are created by artists or professionals such as writers, illustrators, musicians, etc.

This module's root element is *Asset*. Therefore, the purpose of this module is to describe the different kinds of *Assets*, the *Output Actions* that they may participate and the GOs associated to them. Figure 24 presents a diagram with a simplification of this module which has the following properties:

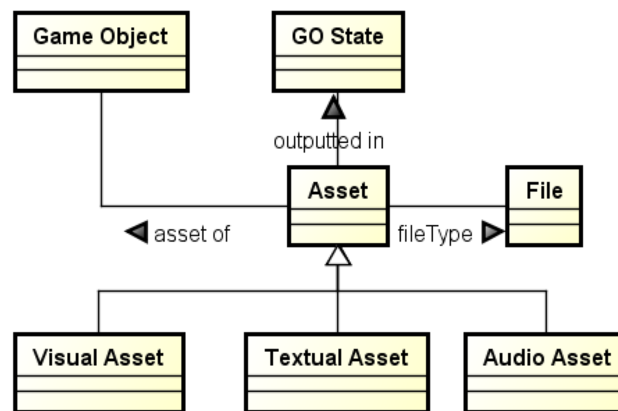

Figure 24 – Asset Module

- *Assets* are associated to a *File* type because the game software needs to open and read them in order to extract the relevant data to manipulate.
- *Assets* are associated to one or more GOs. For example, in fighting games when two players choose the same character, one player has their character model or sprite have different colors. This is called a pallet swap. This happens too in RPGs where stronger enemies appear as pallet swaps of weak enemies.
- *Assets* are associated to one or more *States* of a GO.
- *Assets* are used in an *Output Action* because they are intended to be displayed in some form to the player. Therefore, the *States* which they are associated with are conditions for an *Output Action* (the action performed to display them). **See Axiom 47**.

Assets can be divided in three categories:

- Textual: created by writers, scriptwriters, etc. Includes scripts for non-player character dialogues, item descriptions, option labels, etc.
- Audio: created by musicians, vocalists, etc. Includes music, voice clips, sound effects, etc. They can represent sound of abstract entities (dragons, ghosts, etc.) or real entities (gunfire, a person scream, rain drops, etc.).
- Visual: created by illustrators, 3D modelers, etc. Includes 2D images, 3D models, textures, 2D and 3D animation data, special effects, etc. They can represent part or the totality of abstract concepts such as dragons, flying turtles; and real concepts such as a tree or airplanes.

The following relationships will be left out of the ontology because of the scope of the ontology but it can be included later in an extension:

- *Assets* are created by one or many persons. *Assets* clearly have authors that must be credited for their work on it.
- *Assets* represent a concept. For example, in a video game there can be assets that represent *Mickey Mouse* which is a property of *Disney*.

Both relationships exist because of concerns with copyright and authorship issues since some games use licensed assets such as songs. It is important to identify those issues in the requirement gathering phase of the development. However, it is not on the scope of this ontology to detail the authorship of an *Asset*.

8.9 – Video Game Module

This module's root element is *Video Game* and it extends the *Software* element. Its relationship with the *Hardware* module was already detailed. Therefore, the purpose of this module is to help the developers organize the elements of the *Video Game* in modules and provide general information about it. By organizing the game in modules, it will be possible to create templates of *Video Games* which make reuse possible. Those templates of video games will feature recurring modules. In turn, those modules will feature recurring GOs. A simple example would be of a racing game; it will always feature a race module which feature track objects that contains a starting line and a finish line. Figure 24 presents a diagram with a simplification of this module which has the following properties:

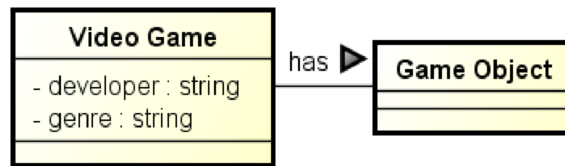

Figure 25 – Video Game Module

- Developer: The organization or person that develops the game.
- Genre: The genres that the video game is part of. Example: racing, action games.
- Since it is a *Software*, a *Video Game* can be run in a number of *Hardware*. In this relationship the developer can include the potential or actual hardware that is necessary to play the game.
- A *Video Game* is composed of *Game Objects*.
- Because a *Video Game* is an extension of a *Software*, it can also be divided in modules. It allows developers to separate the distinct functionalities of the game and the different instances of gameplay the game possesses. A video game can be so simple that it might only need one module but the majority video games are pretty complex. A simple example of *Game Module* elements would be a module dedicated to the *Main Menu* of the game which handles things like starting a new game and continue a game from a saved state. It can be divided further into the *Main Menu* module in *New Game* module and *Load Game* module.
- A *Video Game* can be composed of other *Video Game* elements because it is a *Software*. Some video games can be a collection of games or contain mini-games which would be games inside the game itself and that can be accessed during gameplay. In this case, those elements would be treated as *Game Module* elements. An example of game like this would be the *Professor Layton* series; it is a series of game that has two distinct parts: a visual novel part in which the story unfolds and the player interact with characters and the puzzle part which consists in solving puzzles to advance the story and earn points. The puzzles can range from mathematical problems, logic problems, block pushing puzzles, etc.; each one with its own set of rules and assets. As it can be seen, each puzzle can be a game on its own.

- Examples of extensions are expansions of video games such as *StarCraft: Brood War*. This expansion has the same structure and assets contained in the original *StarCraft* while adding new units for each faction of the game, updating rules in order to balance the game and providing new single player missions.

The developer property is useful because it is very common that developers outsource certain parts of their games to specialized third parties in order to accelerate development of the game. The most common forms of outsourcing are asset production such as 3D models and porting of video games to other platforms (making a game that is originally developed for a console playable in a PC for example). There are cases that portion of the gameplay content of the game is outsourced. A good example is *Deus Ex: Human Revolution* developed by *Eidos Montreal*, where its boss battles were outsourced to *G.R.I.P Entertainment* a third-party developer (WALKER, 2011).

The developer can model a *Video Game* element with a huge degree of freedom and impose restrictions to more specific modules if necessary (impose a module which can only have *Menu* elements for example). This fact supports the objective of not restricting the creativity of the developers. And even if there are modules with duplicate functionalities or isolated modules, the ontology reasoning power can be used to detect those inconsistencies.

To conclude, the *Video Game* module allows the development team to separate the game in parts in order to distribute and quantify the work to be done. It also allows templates to be built and reused in the development of other games.

Chapter 9 – Implementation

As explained in Section 5.2.5, the goal of the implementation activity is to build computable models using ontology implementation languages. The conceptual models detailed in Chapters 7 and 8 are abstract models. In order to get a formal and explicit formalization of this conceptualization, it is necessary to implement the corresponding ontology. The ontology of this work will be formalized and implemented in the OWL 2 language using the Protégé editor. It is recommended that readers not familiar with OWL refer to Appendix A.

In this chapter I will present the implementation process to build the ontology. First, I will present an overview of the implementation process. Second, I will present the individual steps that compose a phase of the implementation process. Finally, I will detail each of the individual phases of the implementation process.

9.1 – Implementation Process Overview

According to the conceptual models detailed in Chapters 7 and 8, the ontology is separated in modules. In addition, the VGDO should follow the Maximum Monotonic Extendibility principle (GRUBER, 1995), which means that the addition of new modules should not modify the already implemented modules. To ensure that the VGDO follow this principle, the ontology implementation process will be divided in phases. Each phase will have the following rules:

- Tightly coupled modules must be implemented in the same phase because a change in one module will affect the other;
- Coupling between modules of different phases must be minimal. In the VGDO, cases of minimal coupling would be extensions of classes and the connection between the internal and external parts of the VGDO (connected through the *Input* and *Output* classes). That way, changes in subsequent phases will not affect already implemented modules of previous phases. There are no guarantees that it will happen in practice, but the chance of it happening is low because subsequent phases are built using the already established and verified definitions of previous phases. Also, there will be no need to verify modules of previous phases again because they are not changed.

- It is possible that during a phase, the conceptualization of the implemented modules must be revised because of inconsistences. It can happen, even though the conceptualization was revised several times before reaching the implementation phase. The scope of the revision is restricted to the modules of the phase and it ends when all inconsistencies are corrected in the verification activity.

The result of the conclusion of a phase is an ontology. Thus, the result of each subsequent phase is an extension of the previous ontology.

To support the implementation process, the tables produced in the conceptualization phase are used as reference for the implementation because they are semi-formal description of the ontology (OWL 2 terms are utilized to fill the tables information). Those tables are located in Appendix B.

9.2 – Implementation Phase Steps

A phase of the ontology implementation process is composed of the following steps:

- Addition of the classes and sub-classes;
- Addition of data properties (binary relation between concept and datatype);
- Addition of object properties (binary relation between two concepts);
- Define class (their value is constant) and instance attributes (their value is different);
- Define relations cardinalities, inverse relations and mathematical properties;
- Add axioms and rules. In OWL most axioms are implemented by specifying disjoint classes or relations, establishing sub-classes and relations restrictions between classes. For example, the *Hardware* and *Software* classes are disjoint with each other; therefore an instance of *Hardware* cannot be an instance of *Software*. Taking it further, I add that *Hardware* is equivalent to an *External Object* that receives *Physical Input* or *Output* while *Software* is equivalent to an *External Object* that receives *Non-Physical Input* or *Output*. If I add that a particular instance of *Hardware* called X receives a *Non-physical Input*, a inconsistency will rise. Since an instance of *Hardware* is also an instance of *External Object* and since X receives a *Non-physical Input*, the reasoner will infer that it is an instance of the *Software* class, however I

already established X as an instance of *Hardware*. Thus, as both *Hardware and Software* are disjoint with each other, it is impossible that X is an instance of both classes;

- Evaluate the ontology;
- Merge ontologies into a new one.

The evaluation process is detailed in Chapter 10. If the evaluation of the resulting ontology is satisfactory, the implementation process goes to a new phase.

9.3 – Implementation Process Phases

Those are the following phases of the ontology implementation process:

- First phase consists in the implementation of the following internal modules: *Game Object*, *Attribute*, *Action*, *Event* and *State*. No extensions will be implemented.
- Second phase consists in the implementation of the following external modules: *External Object*, *Input* and *Output*. No extensions will be implemented. The internal and external modules are connected through the Input and Output modules.
- Third phase consists in implementing the extensions of the already implemented internal modules. *Time* and *Space* modules are not added because of their complexity.
- Fourth phase consists in implementing the *Time* module.
- Fifth phase consists in implementing the *Space* module.
- Sixth phase consists in implementing the *Hardware* and *Software* modules, the physical and non-physical extensions of the *Output* module, all the *Input* module extensions and the *Video Game* module.
- Seventh phase consists in implementing the rest of *Output* module extensions.
- Eighth phase consists in implementing the Asset module.
- Final phase consists in implementing the Player module.

Chapter 10 – Evaluation

According to GARCÍA-GONZÁLEZ (2006), the evaluation activity judges the developed ontologies, software and documentation against a frame of reference. Ontologies should be evaluated before they are used or reused. There are two kinds of evaluation, the technical one, which is carried out by developers, and user's evaluation. In this thesis, only technical evaluation will be performed on the Video Game Development Ontology (VGDO).

In this chapter I will present the evaluation process of the ontology. First, I will present an overview of the evaluation process. Second, I will report the results of the verification activity. Finally, I will detail and report the results of the validation activity.

10.1 – Evaluation Process Overview

The evaluation process of the VGDO is composed of two evaluation activities: verification and validation. Verification refers to building the ontology correctly, that is, ensuring that its definitions implement correctly the requirements or function correctly in the real world. Validation refers to whether the ontology definitions really model the real world for which the ontology was created (GÓMEZ-PÉREZ *et al.*, 1995). The adopted criteria for ontology evaluation are described in Section 5.2.5.

Verification of the ontology is done alongside the implementation process. As it was explained in Chapter 9, in the end of each phase of the implementation, the ontology is evaluated and once it satisfies the evaluation criteria, the phase is concluded and a new phase began anew. Once the implementation is done, a final review of the classes, relations and axioms is done by comparing them to their definitions in the conceptualization. In this evaluation activity, the evaluation criteria are observed in order to perform the necessary corrections and improvements to the ontology. The verification activity is important because the ontology produced is a generic one, i.e., it must serve for a great number of applications, so problems with the ontology must be identified and corrected early in the ontology lifetime.

Validation of the ontology is done when the implementation process concluded. The ontology is guaranteed to have a considerable degree of consistency and conciseness because of the verification activity performed during the implementation. In the validation activity, I will model a gameplay segment of a video game using the VGDO. Like the verification activity, the evaluation criteria are observed in order to perform the necessary corrections and

improvements necessary for the successful modelling of the gameplay segment. Finally, competency questions defined in the specification phase are asked and the ontology must provide accurate answers. Corrections may be done in case the ontology is unable to answer those questions.

10.2 – Ontology Verification Findings

In this section, I will discuss the problems and changes made in each phase of the implementation. It is recommended that the reader use Appendix B as complement for this section.

- **First Phase**

Relations that represent all object properties of a class (*topActionObjectProperty*, for example) were created with the intent to organize the relations because there were many to test. This created the problem that they were inferred by the reasoners when they were not needed to. No changes were made because the benefits of organization outweighed the redundant inferences.

Axiom 2 (An Attribute can only be changed by Actions that belong to the same Game Object) could not be implemented because in OWL 2 instances cannot be differentiated. For example, we have *Action* A1 performed by *Game Object* G1, *Action* A2 performed by *Game Object* G2 and *Attribute* AT1 owned by G1. According to Axiom 2, A2 cannot change AT1 but OWL 2 does not have any formalism to make restriction on individual instances making it impossible to enforce the axiom.

I tried to solve this problem by creating the *canChangeAttribute* relation. It represents the *Attributes* that an *Action* can change. It is a chain property by combining the *isPerformedBy*, the rolification[15] of the *Game Object* class and the *hasGOAttribute* relations. It works by finding the instance of the GO (G1) that performs the *Action* instance (A1) and from that GO instance, the *Attribute* instances (AT1) that the *Action* can change are found. However, assigning *changesAttributes* as a sub-property of it does not work because the reasoner infers that *canChangeAttribute* has the same pair as *changesAttribute*, it does not

[15] http://stackoverflow.com/questions/16989042/owl-2-rolification

restrict the *changesAttribute* property. Fortunately, the *canChangeAttribute* successfully infers the valid Attributes that an Action can change.

hasStatePart cannot have cardinality restrictions because it is a non-simple property (it is transitive). This is restriction was imposed by the reasoner. Therefore, it is impossible to implement Axiom 15a.

No problems were identified in the second phase of the implementation.

- **Third phase**

Data properties such as *datatype* and *initialValue* were not implemented because they are in most part implementation details for when the developers define an actual video game.

Axiom 1 (A GO that contains more than two instances of another GO implies that it has a *Collection* of the contained GO) could not be implemented. OWL 2 does not have formalisms that enable the reasoners to infer automatically that an instance has a relation to an anonymous instance. In this case, the reasoner would have to create an anonymous instance of *Collection* with a relation *collectionOf* with an anonymous instance of the GO.

Created class *SimpleAttribute* that represents the union of a *CompositeAttribute* and *AtomicAttribute* instances to enforce axiom 2a (A Composite Attribute can be only composed by Atomic or Composite Attributes). The reason is that, initially, the *compositePart* relation had as domain *CompositeAttribute* and as range the "*CompositeAttribute* AND *AtomicAttribute*" classes but the range did not represent accurately the union of those classes. Therefore, the *SimpleAttribute* class was created and set as range of the *compositePart* relation.

Reasoner infers relationship *compositePart* again even when the relationship is already asserted. This happens in other relations of other phases too.

- **Fourth phase**

The *hasEquivalence* relation was dropped because it can be modeled using the *State* relation *equivalentState* to model equivalence between *Time*. For example, the *State* "1 minute elapsed" is equivalent to the *State* "60 seconds elapsed".

Created *hasActionState* and *isStateOfAction* relations to model the state of an ongoing *Action*. This was done in order to model the duration of an *Action*.

- **Fifth phase**

Class names cannot begin with numbers in OWL 2, therefore I named the classes *0D*, *1D*, *2D* and *3D Spaces* as: _0DSpace, _1DSpace, _2DSpace and _3DSpace.

Axioms 16 (A *Discrete Space* is composed of *0D Spaces*) and 17 (A *0D Space* must be connected to another *0D Space*) were dropped. Axioms 16 and17 were not implemented because of the open-world assumption, the lack of relations to an instance of a class (absence of information) will not work as restrictions. So an *Discrete Space* without any *0D Spaces* or a *0D Space* not connected to another *0D Space* will not raise inconsistences after the reasoning process.

Space Connection class dropped. It is easier to create subproperties of the relation *isConnectedTo* to describe more specialized connections between *Spaces*.

The class *SpatialAttribute* was created (Axiom 17a) to represent all attributes that determine and change the space of a game. This was done to better define the *Spatial State* and *Spatial Action* classes. Examples are: position, speed, acceleration, shape, etc.

- **Sixth phase**

Inconsistencies that appeared in the ontology were caused by the conceptualization of the Video Game module. Therefore, the conceptualization of this module was revised.

Dropped *Module* class. Instead, added the relations *hasModule* and *isModuleOf* to the *Software* class. Dropped the same relations from the *Video Game* class.

- **Seventh phase**

Inconsistencies that appeared in the ontology were caused by the conceptualization of the Output and Video Output modules. Therefore, the conceptualization of those modules was revised. The resulting changes of this revision were: axioms 37, 38, 43, 44, 45 and 46 changed; added axioms 36, 39 and 40.

Added *affectsOutput* and *outputAffectedByAction* relations to represent *Actions* that affect the *Output* of a game.

Dropped relation *featuredIn* from *Asset* class and its inverse *featuresAsset* from *Output* class

- **Eighth phase**

Because of the revision of the conceptualization, the *Asset* class is not an extension of any class. Before, it was an extension of the *File* class.

Axiom 47 (If a State outputs an Asset it implies that the State is a condition of an Output Action) dropped because of the same reason for dropping Axiom 1.

Included relations *outputtedIn* and *outputsAsset*, they represent the *States* that the *Asset* is sent as an *Output*.

Included relation *fileType*, represent the types of *File* that the *Asset* is available.

- **Last phase**

Axioms 28 and 30 enforced with the creation of the relation *playerSendsInput* and *inputSentByPlayer*. Those relations represent the *Inputs* the *Player* sends through his interaction with the *Hardware*.

Axioms 28 and 33 enforced with the creation of the relation *playerReceivesOutput* and *outputReceivedByPlayer*. Those relations represent the outputs the *Player* receives through his interaction with the *Hardware*.

In the final review, the classes, relations and axioms were revised by comparing them to their definitions in the conceptualization. Also, annotations describing the definitions of the classes and relations of the ontology were added through Protégé. The following problems were identified and changes were made:

- A new axiom was added: an Attribute cannot be possessed by an *External Object* and a *Game Object* simultaneously. Enforced by creating the classes *GOAttribute* and *EOAttribute*, making them disjoint, making those classes the ranges of *hasGOAttribute* and *hasEOAtrribute* respectively.
- The relation *isPartOfOutput* was implemented incorrectly. The relation *hasOutputPart* was created with the Output class as both domain and range as well as relation *isPartOfOutput* assigned as its inverse.
- In the definition of the *Event* class, it is stated that "An Event carries information in the form of Attributes, GOs or a combination of both". However, the relation

hasEventAttribute has only as range the *Attribute* class. For now, I consider it sufficient because an *Attribute* is possessed by a *Game Object* and this fact can be inferred. However, this may change depending on the ontology validation activity.

The ontology verification concluded together with the implementation process. The following conclusions regarding the resulting ontology are:

- The VGDO is consistent so far. Tests were done in all phases to verify all added classes, relations and axioms to check for inferred contradictory knowledge and other consistency problems. By the end of the verification activity no consistency problems were detected. Developing the ontology in Protégé facilitated the process because it detected automatically some consistency problems.

- The VGDO completeness cannot be assessed. This fact is caused by its nature as it is a generic ontology with the purpose to be able to model any kind of video game. Also, it is impossible for an ontology to be complete because new knowledge will be always added to it. It is through the validation activity that the completeness of the VGDO will be assessed as its ability to model video games will be tested. The VGDO will be considered "complete enough" if it can successfully model a video game, i.e., if it has the necessary class, relations and axioms to do so. The VGDO will be truly complete when it can be used to model any type of video game.

- The VGDO is somewhat concise. Tests were done to remove all unnecessary definitions and explicit redundancies between definitions. However, some redundancies can be inferred by the reasoner. For example, the reasoner infers the relation *hasGOAttribute* even if the same relation was asserted directly in the Space class; there was no need to infer something that was known. It has to be investigated if the problem is in the ontology implementation or in the reasoner.

- Cardinality restrictions were dropped because of several problems trying to implement them in non-simple relations (they have properties such as transitivity). However, functional relations were successfully implemented.

10.3 – Ontology Validation

In this section, I will present the ontology validation activity, discuss the results and the changes made to the ontology in order to fix problems found. For this activity, I will use the VGDO to model a gameplay segment of the first stage of *Super Mario Bros.* developed by *Nintendo* for the *Nintendo Entertainment System* (NES) pictured in Figure 26.

The validation activity is divided in five parts:

- Identify the game elements and assign them to their corresponding classes;
- Add important terms found in the identification activity to the VGDO;
- Model the gameplay segment using VGDO through Protégé;
- Compare the result provided by the reasoner and compare them to the competency questions;
- Conclusions of the ontology validation activity.

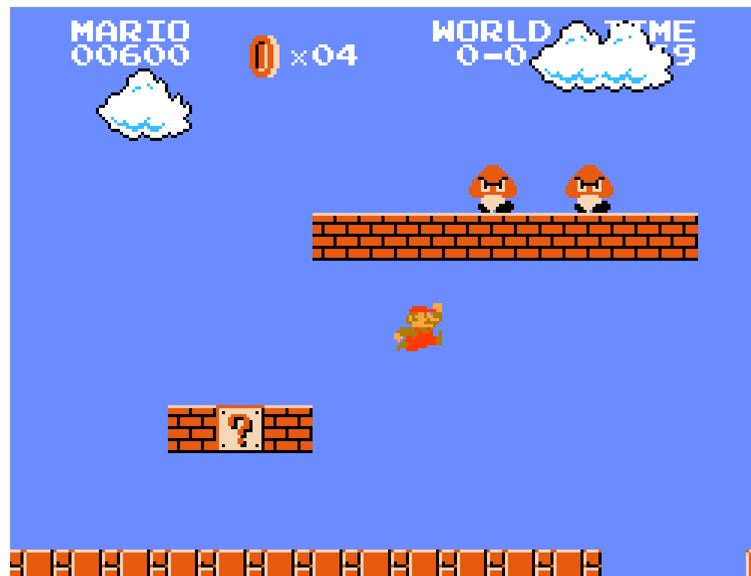

Figure 26 – First stage of *Super Mario Bros.*

10.3.1 – Identification of Game Elements

This activity will function just like a requirements identification or reverse-engineering process. The observed game elements from the first stage of Super Mario Bros. will be identified from the most external to the most internal:

- Players and Hardware necessary to interact with the game;

- The Output the Player receives such as Audio Output and Video Output (Display Space for the player to see and Display Objects featured in it and Audio);
- Assets used in the Game Objects and the States that are sent to the Output;
- Player Objects and the Inputs used to control them;
- Describe the Space logic of the game (the part of gameplay that involves space);
- Describe the Time logic of the game (the part of gameplay that involves time);
- Describe the specific logic (the part of gameplay that does not involve space or time);
- Define the Game Objects by identifying their States, Attributes and specific rules;

The game logic is the most complex part of a video game, so it is divided in space, time and specific (game exclusive) logic. It should be noted that not all elements of that gameplay segment will be identified to keep this validation activity feasible.

10.3.1.1 – Players and Hardware

Super Mario Bros. is played by only one human player. The *Player* needs a *Video Hardware* to see what is happening in the game, an *Audio Hardware* to hear what happens in the game and *Input Hardware* such as a joystick to send *Inputs* to the game. Obviously, the *Player* needs a *Hardware* that can run the *Video Game* but I will omit this detail in this validation activity.

10.3.1.2 – Output and Player View

The *Video Output* of the gameplay segment is composed of three layers:

- The background layer which consists of the stage background art;
- The foreground layer which consists of the interacting game objects such as Mario, Ground, Goombas, etc.
- The interface layer which consists of the score of the player, coin counter, world name and timer.

The background layer is rendered first, followed by the foreground layer and the interface layer is rendered last. The *Audio Output* is composed with two audio streams:

- Background music (BGM) stream that plays all the time during the stage;
- Sound effects (SFX) stream that play short sound clips such as Mario jumping.

Unlike the layers of a *Video Output*, streams of an *Audio Output* can play at the same time. There is only one *Player View* which is a *Display Space* that shows a portion of the stage and moves according to Mario position.

10.3.1.3 – Assets

Table 2 presents some of the identified Visual Assets, the States that they appear and whether they are animated or not. Table 3 presents some of the identified Audio Assets and the States that they are played.

I probably may not have identified all possible assets but this is more than enough. Also, those tables will help in identifying the internal events and inputs that Game Objects handle.

10.3.1.4 – Player Objects and Inputs

The only *Player Object* is Mario. Table 4 presents the pressed *Inputs* that it receives, the subsequent *Actions*, its conditions and outcomes.

10.3.1.5 – Space Logic

The space logic is basically about collisions between Spaces. Table 5 describes the collisions featured in the game.

10.3.1.6 – Time Logic

The time logic in *Super Mario Bros*. is simple. At the beginning of each stage you begin with the timer at 400 seconds and it starts counting down. When it reaches 100 seconds, a warning sound clip plays and the tempo of the background music begins. When the timer reaches 0, Mario loses a life.

10.3.1.7 – Specific Logic

Specific logic is the game logic that does not involve space or time operations between Game Objects. Table 6 describes the rest of the internal events of the game.

Table 2 – Identified Visual Assets

Visual Asset	State	Animated
Mario idle	Mario is on the ground and no inputs made	No
Mario walking	Mario is on the ground and horizontal input made	Yes
Mario jumping	Mario is on the air or (Mario is on the ground and jump button input is pressed)	No
Mario running	Mario is on the ground and running button is pressed	No
Mario crouching	Mario is on the ground and down input is pressed	No
Mario gets a power-up	Mario collides with a power-up	Yes
Mario loses a power-up	Mario has power-up and does not collide on top of Goomba	Yes
Mario loses a life	Mario has no power-up and does not collide on top of Goomba	Yes
Goomba moving	Goomba is not dead	Yes
Stomped Goomba	Mario collides on top of Goomba	No
Goomba flying	Goomba collides with moving block	Yes
Coin	Coin exists on the stage	Yes
Power-up	Power-up exists on the stage	No
Mystery block	Block is a mystery block	Yes
Ground (floor and walls)	Depends on the type of ground	No
Background	Depends on the stage	No
Score of the player	Depends on the score of the player	No
Coin counter	Depends on the number of coins of the player	Yes
World Name	Depends on the stage name	No
Timer	Depends on the time elapsed in the stage	No

Table 3 – Identified Audio Assets

Audio Asset	State
Normal stage BGM	Depends on the stage
Mario jumping	Mario is on the ground and jump button input is pressed
Mario stomping Goomba	Mario collides on top of Goomba
Goomba being hit by a moving block	Goomba collides with moving block
Mario losing a life	Mario has no power-up and does not collide on top of Goomba
Mario hits a mystery block when jumping	Colliding with mysterious block and Mario collides below those blocks while jumping
Mario getting a power-up	Mario collides with a power-up
Mario losing a power-up	Mario has power-up and does not collide on top of Goomba
Mario getting a coin	Mario collides with a coin
Stage timer reaching 100 seconds	-
Urgent stage BGM	Timer is under 100 seconds
Mario reaches the end of stage	Mario collides with flagpole
Mario jumping	Mario is on the ground and jump button input is pressed

Table 4 – Player Inputs

Input	Action	Condition	Outcome
Horizontal direction	Mario walks	Mario is on the ground and not colliding with a wall	Horizontal speed is not equal to zero
Horizontal direction	Mario changes direction on the air	Mario is on the air and not colliding with a wall	Horizontal speed is not equal to zero
Down direction	Mario crouches	Mario is on the ground	Mario crouched state
Horizontal direction and running button	Mario runs	Mario is on the ground and not colliding with a wall	Mario running state and horizontal speed not null
Jump button	Mario jumps	Mario is on the ground	Mario jumping state and vertical speed not null

Table 5 – Space Logic

Collision Handler	Collides With	Action	Condition
Mario vulnerable borders	Goomba	Loses power-up	Mario has power-up
Mario vulnerable borders	Goomba	Mario loses life	Mario has no power-up
Mario bottom border	Goomba	Gets points and bounces	-
Mario bottom border	Nothing	Gravity	-
Mario bottom border	Ground and Blocks	Null vertical velocity	-
Mario lateral borders	Ground and Blocks	Null horizontal velocity	-
Mario top border	Ground and Blocks	Null vertical velocity	Mario is jumping
Mario space	Coin	Increase coin count by 1	-
Mario space	Power-up	Gets power-up	-
Mystery block	Mario top border	Moves. Creates a power-up on the stage	Mario is jumping
Goomba	Mario bottom border	Gets on stomped state	-
Goomba	Blocks	Flies	Block is moving
Power-up and Coin	Mario	Disappears	-

Table 6 – Specific logic

Event	Sender	Handler	Action
Create power-up	Mystery Block	Stage	Create and display power-up object at the top of the block

10.3.1.8 – Game Objects

The identified game objects are: Stage, Mario, Ground, Goomba, Coin, Mystery Block, Unbreakable Block, Brick Block, Power-up and Flagpole. All of them have positions and spaces. Most of its States have been described in the previous tables.

Mario is the most complex game object. It has Attributes and internal rules:

- Speed to measure how much fast it is moving.
- Number of Coins. If it reaches 100 coins, Mario gains a like and the number of coins is reset to zero.
- Point score. It is displayed in the interface.
- Power-up state to determine if Mario has a power-up and which power-up it is.
- Gravity force to determine how fast Mario falls.

Goombas simply have a constant speed attribute that determines which direction they are walking and a point value. Mystery Blocks can contain a number of coins or contain a single power-up. Power-up has an Attribute that point which type of power-up it is. The rest of the objects have no other notable Attributes or internal rules.

With this, I concluded identifying the game elements to describe a gameplay segment of Stage 1 of Super Mario Bros.

10.3.2 – Ontology Extensions

It was necessary to expand the ontology in order to model the gameplay segment in OWL 2 because some of the terms used in the identification activity are not present in the VGDO. The new class, axioms and relations were tested and verified.

The classes *Audio Hardware*, *Video Hardware* and *Mechanical Hardware* were added to the ontology in order to separate which *Hardware* received each kind of *Output* and sent each kind of *Input*. All of them are sub-classes of *Hardware* and are not disjoint of each other because hardware like handheld consoles exists. Those are composed of buttons, speakers and screens.

The relations *inferiorImageLayer* and *superiorImageLayer* were added to the ontology. *Video Output* can be separated in layers that are drawn in top of each other. In Section 10.3.1.2 the *Video Output* layers of *Super Mario Bros.* are detailed.

The class *Collision* was added to the ontology in order to model several types of collisions. Relations *collisionHappensInSpace* and *sendsCollisionEvent* were created to determine the *Space* that contains the two colliding *Spaces* because it is the *Game Object* that

sends the *Collision* event to them. Relations *handlesCollision* and *collisionHandledBy* were created to determine the *Space* that handles the *Collision* and takes an *Action* in response.

Relations *isCollidable* and *collidingSpace* were created to determine the space that causes the collision. Before creating those, the relations *hasEventGO* and *isGOofEvent* were created to determine *Game Objects* that function as a parameter for an *Event*. The relations *isCollidable* and *collidingSpace* are sub-relations of *hasEventGO* and *isGOofEvent*.

The new axiom "two Spaces can only collide if they are contained in the same Space" could not be implemented because of OWL 2 expressivity limitations.

The classes *Visual State* and *Audio State* were added to the ontology in order to describe *GO States* that will feature some kind of *Audio* or *Video Output*. They are disjoint of *Output* class. *Visual States* are *States* that are associated to a *Visual Asset* or are conditions to *Video Output Action*. *Audio States* are *States* that are associated to an *Audio Asset* or are conditions to *Audio Output Action*. As a consequence, the class *Audio Output Action* was added to complete the *Audio State* class.

There was no need to modify any of the VGDO ontology with the exception of having to add disjoint siblings classes to the *Output* class. This fact reinforces the VGDO extendibility. With that, all the necessary class, axioms and relations to model the gameplay segment were added to the ontology.

10.3.3 – Modelling the Gameplay Segment

Part of the gameplay segment was implemented in OWL 2 as part of the validation activity. However, it generated a great number of individuals to implement in OWL 2 because the Protégé interface is unsuitable for a great number of individuals as them and their relations will cause the reasoner to take a long time to conclude the reasoning process. Therefore, the following *Game Objects* were cut along with their interactions: power-ups, coins, blocks, flagpole, coin counter and world name.

By modelling the gameplay segment using the VGDO, flaws in the design can be found because if the VGDO cannot describe the game element then it means either it has a design error or it is lacking an axiom or terms. Also, the results provided by the reasoner from reasoning the implemented slice of the gameplay segment were compared with the

competency questions in Appendix C to check for any irregularities such as facts that should have been inferred but were not.

10.3.4 – Problems in the Modelling Activity

In this section, I will describe the problems that were identified in the modelling activity as well as the solutions for those.

There was no way to model *Input* composed by other *Inputs*. This problem was corrected by creating the class *Simultaneous Input* and the relations *isComposedOfInput* and *composesInput*. The class represents *Physical Input* that is composed of multiple *Physical Inputs* sent simultaneously. The relations represent *Inputs* that compose *Simultaneous Inputs*.

The reasoner was not inferring that a *Game Object* sends *Outputs* contained in a bigger *Output*. The problem was corrected by adding a chain property that lets the reasoner infer that if a *Game Object* sends *Output*, it also sends *Output* that is part of it.

The reasoner was not inferring that a *Game Object* is a *Display Object* if it has a *Visual State*. The problem was corrected by adding the axiom to the *Display Object* class.

Speed, *Shape* and *Acceleration* classes were created, they are extensions of the *Spatial Attribute* class. They were created to be able to model Mario velocity, the shape of the space Mario occupies and the gravity acceleration.

The reasoner was not inferring that a *Game Object* is controlled by a *Player* when it receives the *Player Input*. The problem was corrected by adding a chain property in the relation *controlsGO* that lets the reasoner infer it.

The reasoner was not inferring the intended *Player* of the *Player View*. The problem was corrected by adding relations *playerHasView* and *isViewOfPlayer*.

The reasoner was not inferring that if a *Space* is part of another, it means it is a *Bounded Space*. The problem was corrected by adding the axiom to the *Bounded Space* class.

There were problems modelling collisions between Mario and Goomba. The problem is the fact that there are two sets of collision: the lateral borders of Mario and the bottom border of Mario space. When the lateral borders collide with the Goomba, Mario loses a life. When the bottom border collides, the Goomba is stomped. I have to separate the Goomba collision event in two because it must be known which part of Mario space is colliding.

Therefore, it is impossible to model a *Collision* event that is handled by many sub-spaces of a bigger space and for each sub-space a different action is performed.

The *sendsOutput* relation was causing the reasoner to infer that a *Game Object* that sent *Output* was a *Game Object* that performed an *Action* that had an outcome. This caused inconsistency in the ontology because the chain property meant that any *State* could be an *Output* which was not true because there are *States* that are disjoint of the Output class. The problem was correct by modifying the chain property: a *Game Object* that sent *Output* was a *Game Object* that performed an *Action* that had an *Output* as an outcome. To this end, the relations *generatesOutput* and *outputGeneratedByAction* were created.

10.3.5 – Ontology Validation Conclusion

The main challenge in the validation activity was the limitations imposed by the Protégé editor. The modelling activity was considerable set back because the scope of gameplay segment was reduced. The Protégé editor is not suitable for the development of complex ontologies that need large knowledge bases to be tested. A more suitable application must be used if more large-scale knowledge bases are needed and to better test the VGDO modelling capacity.

Even though the scope of the modelling activity was reduced, the activity was instrumental to extend and find problems in the ontology. By modelling different types of game, the ontology surely will become more complete and consistent.

It is easy to conclude that the VGDO is far from complete because of the number of terms added to it during the validation activity. However, what is important is the VGDO capacity to be extended without compromising its existing class and relations. The extendibility of the VGDO was easily validated as many of the added terms were easily integrated into existing classes as the result

After the validation activity, the VGDO is more consistent and cohesive because of the many inconsistencies that were found and corrected. This reinforces the importance of an evaluation activity in the production of a reliable ontology.

Finally, the VGDO helped in the identification of the game elements as they were categorized in classes of the ontology. This shows that the VGDO can assist in the requirements identification process.

Chapter 11 – Conclusion

In this chapter, the final considerations regarding the dissertation, comparison to other works, its limitations, its contributions and potential future works.

11.1 – Final Considerations

This thesis proposed the creation of the Video Game Development Ontology (VGDO), a generic ontology describing the video game domain, with the objective of providing a common vocabulary and assisting in the transition between preproduction and production phases of the game development process by helping the identification of technical requirements (which would be implied knowledge) in the game design (GDDs, concept art). Thus, it makes the gathering of requirements more accurate and reliable and, in consequence, mitigates several problems found in the game development process.

In order to create the VGDO, knowledge about the video game domain was necessary, thus a knowledge acquisition activity was performed. Research was done to pinpoint the most essential elements found in games and video games by analyzing the available literature on game design and video game development; followed by the analysis of existing knowledge representations, especially ontologies, of games and video games.

To build the ontology correctly, the METHONTOLOGY ontology building methodology was chosen. Thus, the ontology construction was done in four main phases: specification, conceptualization, implementation and evaluation. In the specification phase, an overview of the VGDO explaining the motivation, intended users, scopes and other relevant information was provided. In the conceptualization phase, the knowledge acquired is organized and structured in modules. In the implementation phase, an implementation process was outlined and the ontology was implemented in OWL 2 following this process.

Finally, the evaluation phase consisted of two distinct activities: verification and validation. In both of these activities, the implemented ontology was evaluated according to established criteria and changes were made to fix errors and improve the ontology. The validation activity was of vital importance as it evaluated the VGDO capacity of modeling gameplay of video game and its extendibility.

11.2– Comparison to Other Game Ontologies

In this section, I will compare the VGDO in relation to GCM (Game Content Model) and GOP (Game Ontology Project) because they are the most generic game ontologies available. The other ontologies cover specific parts of the domain of video games.

GOP does not follow an ontology construction methodology. GCM follows NOY & MCGUINNESS (2001) methodology. VGDO follows METHONTOLOGY (FERNÁNDEZ-LÓPEZ *et al.*, 1997)..

GOP primary function is to serve as a framework for exploring research questions related to games and gameplay; it also contributes to a vocabulary for describing, analyzing and critiquing games (ZAGAL *et al.*, 2005). GCM is used to document the design specification of a computer game and will be the model for building other game models (TANG & HANNEGHAN, 2011). VGDO was created with the purpose to facilitate the transition of pre-production to production by being used to assist in the identification of requirements.

GOP scope covers important structural elements of games, the top level of the ontology consists of five elements: interface, rules, goals, entities, and entity manipulation (ZAGAL *et al.*, 2005). GCM scope covers the game concepts used in documentation of role-playing and simulation game genres because the authors believe they are more suitable for use in the context of education and training compared to other game genres (TANG & HANNEGHAN, 2011). VGDO scope covers the most basic elements that compose a video game.

GOP is not formalized in any ontology representation language, it was only a taxonomy of concepts. GCM is formalized in XML but details of its implementation are absent (axioms, for example), so it is hard to evaluate its structure. VGDO is formalized in OWL 2 and details of its implementation, such as axioms, are available.

GOP documentation was previously available on the internet (wiki) but was abandoned because of lack of use. GCM has no resources available online besides its paper. I made the VGDO implementation in OWL 2 available online. Download link: https://www.dropbox.com/s/jfm9qoftl6cfahi/video_game_development_ontology.7z?dl=0
Password to open the compressed archive: **VGDO_COPPE_glauco**.

11.3 – Limitations

In this section, I discuss the several limitations that were found throughout the development of the thesis.

The first was the lack of a common vocabulary used by video game developers and game designers. This indicated a lack of consensus in the video game domain which made acquiring accurate knowledge even more difficult because it is a multidisciplinary domain, i.e., a really complex domain.

Second, none of the video game ontologies were reused in the construction of the VGDO, which made its construction difficult. The reason is because several ontologies were too specific and the ontologies that were generic were not implemented in OWL.

Another problem that rises from the lack of reuse is that the VGDO violates one of SMITH (2006) principles: re-using available resources. While the available game ontologies were inadequate for reuse, the Space and Time modules could be built reusing available upper level ontologies. However, I avoided reusing such ontologies because of the following reasons:

- Those ontologies may use a vocabulary far removed from the video game development domain. There is a risk that their terms and definitions will cause confusion in the video game development team making part of the VGDO hard to use;
- Integrating those ontologies would cost time because they would need to be analyzed, the relevant terms and axioms would be selected (pruning) and that portion would be formalized in the appropriate ontology language (they may be formalized in another). In short, a careful study must be made in order to reuse correctly an existing ontology and there was not enough time to do so.

Third, the VGDO had to have its design revised and some of its axioms were not implemented and validated because of OWL 2 limited expressivity. This happened because of my lack of experience with OWL 2 as I elaborated axioms without knowledge of OWL 2 limitations. Also, some axioms were not implemented because of reasoner limitations. As an example, the *Visual State* class is a *State* that has a *Visual Asset* associated to it. The user can create an individual of the *Visual State* class without associating a *Visual Asset* to it. However, the reasoner does not point out any inconsistency in the assertion.

Fourth, there was only enough time to perform a technical evaluation on the VGDO making a user evaluation impossible to be done. A user evaluation would require significant time and preparation because it requires gathering adequate participants, preparing interviews, having a period of time for the users to test the ontology, organize the data provided by the users, draw conclusions from the data and make the necessary adjustments for the ontology. As a consequence it makes the VGDO a not consensual ontology which reduces the reliability and usefulness of the VGDO because the intended audience did not get to use it to evaluate if it brings any benefits. It does not accurately represent a consensus about the video game domain even though the knowledge is originated from the video game domain literature.

Fifth, the technical evaluation may not be enough because the video game domain is immense with different kinds of games. The gameplay modelled in the validation activity of the evaluation process is just a tiny part of this domain. Therefore, it is necessary to model other kinds of games.

Finally, there were no modelling tools suitable for modelling a gameplay segment with VGDO because of the high number of elements identified. Protégé had several limitations that made modelling such large number difficult. For example, its interface is incapable of separating individuals by class meaning that individuals of the *State* class could be mixed with ones of the *Game Object* class which caused confusion. Another example, is the lack of visual modelling tools (it has a graphics tool but it only generates graphs from the created ontology and knowledge base).

11.4 – Contributions

This thesis provides the following contributions: knowledge about video game development; a common vocabulary to be used in the video game development process; improvement of the requirements identification activity; support for analysis and reverse-engineering of video games; a generic ontology that can model video games or parts of them and validation of the hypothesis.

To create the VGDO, a knowledge acquisition activity was done in order to obtain knowledge about the video game domain. Chapter 3 presents knowledge obtained about video games while Chapters 6, 7 and 8 organize and structure of the knowledge obtained. The

reader benefits from those chapters as he learns the complexity of video game development and of a video game structure.

The vocabulary provided by the VGDO allows developers to share information about artifacts that are being developed, created and used in the video game development process. The vocabulary can also help in the standardization of video game documentation by using VGDO. Documentation that uses VGDO terms will surely improve the requirements identification activity in the development process. The vocabulary can also be used to assist in the analysis of existing video games.

The ontology validation activity helped display both the VGDO ability to model a gameplay segment and the VGDO ability to improve the requirements identification activity. Requirements were easily identified from the picture of a gameplay moment of *Super Mario Bros.*, as things from the picture were easily decomposed as what the game objects were, their assets, their actions, etc. It can be also be seen as a tool for reverse-engineering because a developer can simply watch a video game and decompose the gameplay segments using VGDO terms.

The validation of the hypothesis that it is possible to identify knowledge, from other domains present in the game development process, which is hidden implicitly in the game designer knowledge (can be in the form of images, videos, documents) using an ontology was possible because of the validation activity. During the validation activity, game elements were successfully identified using VGDO terms when analyzing the gameplay of a game. They were classified in groups (Sections 10.3.1.1 to 10.3.1.7) and tables were made in order to organize the knowledge found (each column was represented by instances of a class of the ontology). It could be said that Section 10.3.1 was a short Game Design Document but with information such as hardware, assets, attributes and their values, game object behavior. In short, it had game designer knowledge as well as knowledge from other domains of the development team such as programming and art.

11.5 – Future Work

In this section, I discuss the potential improvements of the VGDO and future works that use it, some that can be solutions of the limitations already described.

One of the main limitations that held back the development of VGDO was OWL 2 limited expressivity. A solution would be to add Semantic Web Rule Language[16] (SWRL) rules to the VGDO. SWRL rules can express rules and restrictions otherwise impossible in OWL 2.

The VGDO can be extended in several directions because of its generic nature. For example, the *Hardware* module can be extended to describe existing hardware helping developers to choose adequate hardware and find the necessary information for the development process. Another example is creating an occupation module, which links a development occupation with the creation of a specific *Asset*. This knowledge can be used to estimate the necessary manpower for the development of a video game.

There are three interesting future works for the VGDO: user evaluation, automatic code generation and an application that uses it, especially one for modelling video games. The user evaluation of the VGDO is important for its validation, as it was already explained an ontology must be consensual, .i.e., the users of the domain must agree on the knowledge representation. By receiving user feedback, the ontology can be improved in ways not possible in a technical evaluation.

Automatic code generation is useful for prototyping games. The *Game Content Model* (TANG & HANNEGHAN, 2011) is an example of ontology used for that purpose. Using the ontology as a model and following the Model Driven Architecture methodology it is possible to create video game prototypes composed of software artifacts generated from ontology elements. The VGDO ability of modelling video games would also be further evaluated and improved.

The validation activity demonstrated that the VGDO by itself is unsuitable for use outside the requirement identification activity of the video game development process because there are not adequate tools to use it for modelling. Thus, for the developers it is a better investment to use available tools (many that allow fast prototyping) and learn their technologies instead of spending time learning OWL 2 and its intricacies. Developers will most likely want to use an application that hides those intricacies, streamlines the ontology

[16] http://www.w3.org/Submission/SWRL/

modelling process and benefits from the reasoning abilities of the ontology by pointing out modelling errors and inconsistencies in the design of the video game. Another type of useful application is a knowledge repository of the ontology that can be browsed and modified with ease, just like a wiki.

Bibliographic References

ADAMS, E., DORMANS, J., 2012, *Game Mechanics: Advanced Game Design*. 1 ed, Berkeley, CA, USA, New Riders.

AHMED, E., July 2008, "Use of Ontologies in Software Engineering". *17th International Conference on Software Engineering and Data Engineering (SEDE-2008)*, Los Angeles, California, USA.

ARAÚJO, M., ROQUE, L., September 2009, "Modeling Games with Petri Nets." In: *DiGRA '09 - Proceedings of the 2009 DiGRA International Conference: Breaking New Ground: Innovation in Games, Play, Practice and Theory*. Brunel University.

AßMANN, U., ZSCHALER, S., WAGNER, G., 2006, "Ontologies, Meta-Models, and the Model-Driven Paradigm". In: CALERO, C., RUIZ, F., PIATTINI, M. (eds), *Ontologies for Software Engineering and Software Technology*, Chapter 9, Springer Berlin Heidelberg.

AVEDON, E. M., SUTTON-SMITH, B., 1971, *The Study of Games*.

BERNERS-LEE, T., 2000, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. 1 ed, San Francisco, HarperBusiness.

BERNERS-LEE, T., HENDLER, J., LASSILA, O., 2001, "The Semantic Web". *Scientific American*, v. 284, n. 5, pp. 34–43.

BETHKE, E., 2003, *Game Development and Production*. Plano, Texas, Wordware Publishing Inc.

BJÖRK, S., LUNDGREN, S., HOLOPAINEN, J., November 2003, "Game Design Patterns". In: *Proceedings of Level Up: Digital Games Research Conference 2003*.

BLOW, J., 2004, "Game Development: Harder Than You Think". *ACM Queue*, v. 1, n. 10, pp. 28–37.

BORST, W. N., 1997, *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. Ph.D. Thesis, Enschede, Universiteit Twente.

BRACHMAN, R., LEVESQUE, H., 2004, *Knowledge Representation and Reasoning*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc.

BRATHWAITE, B., SCHREIBER, I., 2008, *Challenges for Game Designers*. 1 ed. Boston, MA, USA, Charles River Media.

BREYER, F., MOURA, L., CAVALCANTI, G., et al., October 2009, "Pesquisa de Jogos Similares Como Fonte de Conhecimento Para Construção de Uma Ontologia de Jogos de Simulação Casuais". In: *VIII Brazilian Symposium on Computer Games and Digital Entertainment*, Rio de Janeiro, RJ, Brazil.

BROM, C., and ABONYI A., April 2006, "Petri-Nets for Game Plot". *AISB'06: Adaptation in Artificial and Biological Systems*, University of Bristol, Bristol, England.

CALLELE, D., NEUFELD, E., Schneider, K., August 2005, "Requirements Engineering and the Creative Process in the Video Game Industry". In: *13th IEEE International Conference on Requirements Engineering, 2005. Proceedings*, pp. 240–250, Paris, France.

———, August 2011, “A Report on Select Research Opportunities in Requirements Engineering for Videogame Development”. In: *2011 Fourth International Workshop on Multimedia and Enjoyable Requirements Engineering - Beyond Mere Descriptions and with More Fun and Games (MERE)*, pp. 26–33, Trento, Italy.

CARDENAS, Y. G,. 2014, *Modelo de Ontologia Para Representação de Jogos Digitais de Disseminação Do Conhecimento*. M.Sc Dissertation, Universidade Federal de Santa Catarina.

CHAN, J.T.C., YUEN, W.Y.F., November 2008, “Digital Game Ontology: Semantic Web Approach on Enhancing Game Studies”. In: *9th International Conference on Computer-Aided Industrial Design and Conceptual Design. 2008. CAID/CD 2008*, pp. 425–429, Kunming, China.

CHANDRASEKARAN, B., JOSEPHSON, J.R., BENJAMINS, V.R., 1999, “What Are Ontologies, and Why Do We Need Them?”. *IEEE Intelligent Systems and Their Applications*, v. 14, n. 1, pp. 20–26.

CHURCH, D. 1999, *Formal Abstract Design Tools*. Available at: <http://www.gamasutra.com/view/feature/3357/formal_abstract_design_tools.php>. Accessed: 2015-11-05 14:22:54

CONGDON, S., 2008, *Bridging the Gap: Interdisciplinary Documentation for Video Game Design*. B.A. Thesis, Algoma University.

COOK, D., 2006, *Lost Garden: GameInnovation.org*. Available at: <http://www.lostgarden.com/2006/04/gameinnovationorg.html>. Accessed: 2015-06-25 19:44:15

CORCHO, O., FERNÁNDEZ-LÓPEZ M., GÓMEZ-PÉREZ A., 2003, “Methodologies, Tools and Languages for Building Ontologies: Where Is Their Meeting Point?”. *Data & Knowledge Engineering*, v. 46, n. 1, pp. 41–64.

———, 2006, “Ontological Engineering: Principles, Methods, Tools and Languages”. In: CALERO, C., RUIZ, F., PIATTINI, M. (eds), *Ontologies for Software Engineering and Software Technology*, Chapter 1, Springer Berlin Heidelberg.

CORCHO, O., FERNÁNDEZ-LÓPEZ, M., GÓMEZ-PÉREZ, A., et al., 2005, “Building Legal Ontologies with METHONTOLOGY and WebODE”. In: *Law and the Semantic Web*, v. 3369 , pp. 142–57. *Lecture Notes in Computer Science*, Springer Berlin Heidelberg.

COSTIKYAN, G., June 2002, “I Have No Words & I Must Design: Toward a Critical Vocabulary for Games”. In: *Computer Games and Digital Cultures Conference Proceedings*, Tampere, Finland.

CRAWFORD, C., 1984, *The Art of Computer Game Design*. Berkeley, CA, USA, Osborne/McGraw-Hill.

DORMANS, J., 2009, *Machinations Elemental Feedback Structures for Game Design*. Available at: <http://www.jorisdormans.nl/article.php?ref=machinations>. Accessed: 2015-06-25 14:15:17

———, 2012a, *Engineering Emergence: Applied Theory for Game Design*. Ph.D. Thesis, Amsterdam University of Applied Sciences.

———, 2012b, “The Effectiveness and Efficiency of Model Driven Game Design”. In: *Entertainment Computing - ICEC 2012*, v. 7522, pp. 542–48., *Lecture Notes in Computer Science*,Springer Berlin Heidelberg.

ELIAS, G. S., GARFIELD, R., GUTSCHERA, K. R., et al., 2012, *Characteristics of Games*. Cambridge, MA, The MIT Press.

ENGLAND, L., 2014, *Types of Designers*. Available at: <http://www.lizengland.com/blog/2014/06/types-of-designers/>. Acessed: 2015-05-25 17:51:29

ESA, 2015, “Industry Facts”. Available at: <http://www.theesa.com/about-esa/industry-facts/>. Accessed: 2015-05-04 22:33:49

FENSEL, D., 2003, *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. 2 ed, Secaucus, NJ, USA, Springer-Verlag New York Inc.

FERNÁNDEZ-LÓPEZ, M., GÓMEZ-PÉREZ, A., 2002, “Overview and Analysis of Methodologies for Building Ontologies”. *The Knowlege Engineering Review*, v. 17, n. 2, pp. 129–156.

FERNÁNDEZ-LÓPEZ, M., GÓMEZ-PÉREZ, A., Juristo, N., March 1997, “METHONTOLOGY: From Ontological Art Towards Ontological Engineering”. In: *Proceedings of the Ontological Engineering AAAI-97 Spring Symposium Series*, Palo Alto, California.

FLODD, K., 2003, *Game Unified Process*. Available at: <http://www.gamedev.net/page/resources/_/technical/general-programming/game-unified-process-r1940>. Acessed: 2015-05-24 20:42:52

FLYNT, J. P., SALEM, O., 2004, *Software Engineering for Game Developers*. 1 ed., Boston, MA, Course Technology PTR.

FULLERTON, T., 2014, *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. 3 ed., Boca Raton, A K Peters/CRC Press.

FURTADO, A. W. B., SANTOS, A. L. M.,October 2006, “Using Domain-Specific Modeling towards Computer Games Development Industrialization”. In: *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM’06)*, University of Jyväskylä, Finland.

GANDON, F. L., 2010, “Ontologies in Computer Science: These New ‘Software Components’ of Our Information Systems”. In: GARGOURI, F., JAZIRI, W. (eds), *Ontology Theory, Management and Design: Advanced Tools and Models*, Chapter 1, Hershey, PA, USA, IGI Global.

GARCÍA-GONZÁLEZ, R., 2006, *A Semantic Web Approach to Digital Rights Management*. Ph.D. Thesis, Barcelona, Universitat Pompeu Fabra, Departament de Tecnologia.

GAŠEVIĆ, D., KAVIANI, N., MILANOVIĆ, M., 2009, “Ontologies and Software Engineering”. In: *Handbook on Ontologies*, pp. 593–615, *International Handbooks on Information Systems*, Springer Berlin Heidelberg.

GAŠEVIC, D., DJURIC, D., DEVEDŽIC, V., 2009a, "Knowledge Representation". In: *Model Driven Engineering and Ontology Development*, 2 ed, Chapter 1, Springer Berlin Heidelberg.

———, 2009b, "Ontologies". In: *Model Driven Engineering and Ontology Development*, 2 ed, Chapter 2, Springer Berlin Heidelberg.

———, 2009c, "The Semantic Web". In: *Model Driven Engineering and Ontology Development*, 2 ed, Chapter 3, Springer Berlin Heidelberg.

GÓMEZ-PÉREZ, A., BENJAMINS R., August 1999, "Overview of Knowledge Sharing and Reuse Components: Ontologies and Problem-Solving Methods". In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99) Workshop KRR5: Ontologies and Problem-Solving Methods: Lesson Learned and Future Trends*, v. 18. Stockholm, Sweden.

GÓMEZ-PÉREZ, A., FERNANDEZ, M., VICENTE, A. de, August 1996, "Towards a Method to Conceptualize Domain Ontologies". In: *Proceedings Workshop: Ontological Engineering*, pp. 41–51, Budapest, Romania.

GÓMEZ-PÉREZ, A., JURISTO, N., PAZOS, J., 1995, "Evaluation and Assessment of the Knowledge Sharing Technology". In: *Towards Very Large Knowledge Bases*, pp. 289–296, The Netherlands, IOS Press.

GREGORY, J., 2014, *Game Engine Architecture*. 2 ed, Boca Raton, A K Peters/CRC Press.

GRUBER, T. R., 1993, "A Translation Approach to Portable Ontology Specifications". *Knowledge Acquisition*, v. 5, n. 2, pp. 199–220.

———, 1995, "Toward Principles for the Design of Ontologies Used for Knowledge Sharing". *International Journal of Human-Computer Studies*, v. 43, n. 5-6, pp. 907–928.

GRÜNINGER, M., FOX, M. S., April 1995. "Methodology for the Design and Evaluation of Ontologies". In: *IJCAI'95, Workshop on Basic Ontological Issues in Knowledge Sharing*.

GRÜNVOGEL, S., 2005, "Formal Models and Game Design". *Game Studies the International Journal of Computer Game Research*, v. 5, n. 1.

GUARINO, N., 1998, "Formal Ontology and Information Systems". In: *Proceedings of Formal Ontology in Information System*, pp. 3–15, IOS Press.

GUIZZARDI, G., 2005, *Ontological Foundations for Structural Conceptual Models*. Ph.D. Thesis, Enschede, CTIT, Centre for Telematics and Information Technology.

GUIZZARDI, G., FALBO, R. A., FILHO, P., et al., 2002, "Using Objects and Patterns to Implement Domain Ontologies". *Journal of the Brazilian Computer Society*, v. 8, n. 1, pp. 43–56.

HALLBERG, N., JUNGERT, E., PILEMALM, S., 2014, "Ontology for Systems Development". *International Journal of Software Engineering and Knowledge Engineering*, v. 24, n. 3, pp. 329–345.

HAPPEL, H., SEEDORF, S., November 2006, "Applications of Ontologies in Software Engineering". *International Workshop on Semantic Web Enabled Software Engineering*

(SWESE'06), held at the 5th International Semantic Web Conference (ISWC 2006), Athens, GA, USA.

HUNICKE, R., LEBLANC, M., ZUBEK, R., July 2004, “MDA: A Formal Approach to Game Design and Game Research”. In: *Proceeding of the AAAI-04 Workshop on Challenges in Game AI*, San Jose, California.

JÄRVINEN, A., November 2003, “Making and Breaking Games: A Typology of Rules”. In: *DiGRA ’03 - Proceedings of the 2003 DiGRA International Conference: Level Up*, Utrecht, The Netherlands.

———, September 2007, “Introducing Applied Ludology: Hands-on Methods for Game Studies”. In: *Proceedings of DiGRA 2007 Conference*, Tokyo, Japan.

JAZIRI, W.,GARGOURI, F., 2010, “Ontology Theory, Management and Design: An Overview and Future Directions”. In: GARGOURI, F., JAZIRI, W. (eds), *Ontology Theory, Management and Design: Advanced Tools and Models*, Chapter 11, Hershey, PA, USA, IGI Global.

JUUL, J., November 2003, “The Game, the Player, the World: Looking for a Heart of Gameness”. In: *DiGRA ’03 - Proceedings of the 2003 DiGRA International Conference: Level Up*, Utrecht, The Netherlands.

KANODE, C.M., HADDAD, H.M., April 2009, “Software Engineering Challenges in Game Development”. In: *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*, *ITNG ’09*, pp. 260–265, Las Vegas, Nevada, USA.

KASURINEN, J., MAGLYAS, A., SMOLANDER, K., 2014, “Is Requirements Engineering Useless in Game Development?”. In: *Requirements Engineering: Foundation for Software Quality*, v. 8396, pp. 1–16, *Lecture Notes in Computer Science*, Springer International Publishing.

KNUBLAUCH, H., OBERLE, D., TETLOW, P., et al., 2006, *A Semantic Web Primer for Object-Oriented Software Developers*. Available at: <http://www.w3.org/TR/sw-oosd-primer/>. Acessed: 2015-05-03 15:06:56.

KREIMEIER, B., 2002, *The Case For Game Design Patterns*. Available at: <http://www.gamasutra.com/view/feature/132649/the_case_for_game_design_patterns.php>. Acessed: 2015-06-24 00:17:21.

LEÓN , A.C. Z., SÁNCHEZ, L. A., November 2010, “An Ontology for Mobile Video Games”. In: *MICAI '10 Proceedings of the 2010 Ninth Mexican International Conference on Artificial Intelligence*, pp. 154–159, Pachuca, Mexico.

LEWIS, J. P., MCGUIRE, M.,FOX, P., 2007, “Mapping the Mental Space of Game Genres”. In: *Proceedings of the 2007 ACM SIGGRAPH Symposium on Video Games*, pp. 103–108, *Sandbox ’07*, New York, NY, USA, ACM.

LING, Y., HUAMAO, G., WANG, X., December 2008, “An Ontology-Based Development Framework for Edutainments”. In: *ISBIM '08 Proceedings of the 2008 International Seminar on Business and Information Management*, v. 1, pp. 343–346, Wuhan, China.

LING, Y., HUA-MAO, G., WANG, X., et al., August 2007, “A Fuzzy Ontology and Its Application to Computer Games”. In: *FSKD '07 Proceedings of the Fourth International*

Conference on Fuzzy Systems and Knowledge Discovery, v 4, pp 442–46, Haikou, Hainan, China.

LLANSÓ, D., GÓMEZ-MARTÍN, M. A., Gómez-Martín, P. P., et al., 2011a, “Explicit Domain Modelling in Video Games”. In: *Proceedings of the 6th International Conference on Foundations of Digital Games*, pp. 99–106, *FDG ’11,* New York, NY, USA, ACM.

———, 2011b, “Knowledge Guided Development of Videogames”. *Artificial Intelligence in the Game Design Process, Papers from the 2011 AIIDE Workshop*, Stanford, California

MACHADO, A. F. V., AMARAL, F. N., CLUA, E., 2009, *A Trivial Study Case of the Application of Ontologies in Electronic Games*.

MALCHER, F., NEVES, A., FALCÃO, L., October 2009, “Aplicação Do Game Ontology Project No Processo de Análise de Similares Para Design de Jogos.” *VIII Brazilian Symposium on Computer Games and Digital Entertainment*, Rio de Janeiro, RJ, Brazil.

MIZOGUCHI, R., 2001, “Ontological Engineering: Foundation of the Next Generation Knowledge Processing”. In: *Web Intelligence: Research and Development*, v. 2198, pp. 44–57, *Lecture Notes in Computer Science*, Springer Berlin Heidelberg.

NECHES, R., FIKES, R., FININ, T., et al., 1991, “Enabling Technology for Knowledge Sharing”. *AI Magazine*, v. 12, n. 3, pp. 36–56.

NELSON, M., MATEAS, M., 2007, “Towards Automated Game Design”. In: *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, v. 4733, pp. 626–637, *Lecture Notes in Computer Science*, Springer Berlin Heidelberg.

NIESENHAUS, J., LOHMANN, S., 2009, “Marrying Game Development with Knowledge Management: Challenges and Potentials”. In: *Networked Knowledge - Networked Media*, v. 221, pp. 321–36, *Studies in Computational Intelligence*, Springer Berlin Heidelberg.

NINTENDO, 2015, *Iwata Asks : Xenoblade Chronicles X : Creating a Whole Planet*. Available at: <http://iwataasks.nintendo.com/interviews/#/wiiu/xenoblade-chronicles-x/0/0>. Acessed: 2015-06-14 15:15:47

NOGUERAS-ISO, J., LACASTA, J., TELLER, J., et al., 2010, “Ontology Learning from Thesauri: An Experience in the Urban Domain”. In: GARGOURI, F., JAZIRI, W. (eds), *Ontology Theory, Management and Design: Advanced Tools and Models*, Chapter 11, Hershey, PA, USA, IGI Global.

NOY, N. F., MCGUINNESS , D. L., 2001, *Ontology Development 101: A Guide to Creating Your First Ontology*. Technical Report SMI-2001-0880, Stanford Knowledge Systems Laboratory.

PETRILLO, F., PIMENTA, M., TRINDADE, F., et a., 2008, “Houston, We Have a Problem...: A Survey of Actual Problems in Computer Games Development”. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*, pp. 707–711, *SAC ’08*, New York, NY, USA, ACM.

RAIES, K., KHEMAJA, M., 2014, “Towards Gameplay Ontology for Game Based Learning System Design Process Monitoring”. In: *Proceedings of the Second International Conference on Technological Ecosystems for Enhancing Multiculturality*, pp. 255–260, *TEEM ’14*, New York, NY, USA, ACM.

REYNO, E. M., CUBEL, J. Á. C., 2008, “Model-Driven Game Development: 2d Platform Game Prototyping”. In: *Proceedings of the GAME ON Conference, 2008*, pp. 5-7.

———, 2009a, “Automatic Prototyping in Model-Driven Game Development”. *Computers in Entertainment (CIE) - SPECIAL ISSUE: Media Arts and Games (Part II)*,v. 7, n. 2, article 29.

———, 2009b, “A Platform-Independent Model for Videogame Gameplay Specification”. In: *DiGRA ’09 - Proceedings of the 2009 DiGRA International Conference: Breaking New Ground: Innovation in Games, Play, Practice and Theory*, Brunel University, West London, UK.

ROGERS, S., 2013, *Level Up: Um Guia Para o Design de Grandes Jogos*, 1 ed, Edgard Blucher.

ROMAN, M., SANDU, I., BURAGA, S., 2011, “OWL-Based Modeling of RPG Games”. *Studia Universitatis Babes-Bolyai, Informatica*, v. 56, n. 3.

RUIZ, F., HILERA, J. R., 2006, “Using Ontologies in Software Engineering and Technology”. In: CALERO, C., RUIZ, F., PIATTINI, M. (eds), *Ontologies for Software Engineering and Software Technology*, Chapter 1, Springer Berlin Heidelberg.

SALAZAR, M.G., MITRE, H.A., OLALDE, C.L., et al., July 2012, “Proposal of Game Design Document from Software Engineering Requirements Perspective”. In: *CGAMES '12 Proceedings of the 2012 17th International Conference on Computer Games: AI, Animation, Mobile, Interactive Multimedia, Educational & Serious Games (CGAMES)*, pp. 81–85, Louisville, KY, USA.

SARINHO, V., APOLINÁRIO, A., November 2008, “A Feature Model Proposal for Computer Games Design.” In: *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, pp. 54–63, Belo Horizonte, MG, Brazil.

SCHAUL, T., 2013. “A Video Game Description Language for Model-Based or Interactive Learning”. In: *Proceedings of the IEEE Conference on Computational Intelligence in Games*, Niagara Falls, IEEE Press.

———, 2014, “An Extensible Description Language for Video Games”. *IEEE Transactions on Computational Intelligence and AI in Games*, v. 6, n. 4, pp. 325–331.

SCHELL, J., 2014, *The Art of Game Design: A Book of Lenses*, 2 ed, Boca Raton, A K Peters/CRC Press.

SEIDEWITZ, E., 2003, “What Models Mean”. *IEEE Software*, v. 20, n. 5, pp. 26–32.

SEQUEDA, J., 2012, “Introduction to: Open World Assumption vs Closed World Assumption”. Available at: <http://www.dataversity.net/introduction-to-open-world-assumption-vs-closed-world-assumption/>. Acessed: 2015-11-01 21:20:35.

SMITH, B., 2006. “Against Idiosyncrasy in Ontology Development.” In: *Proceedings of the 2006 Conference on Formal Ontology in Information Systems: Proceedings of the Fourth International Conference (FOIS 2006)*, 15–26. Amsterdam, The Netherlands, IOS Press.

SPYNS, P., MEERSMAN, R., JARRAR, M., 2002, “Data Modelling Versus Ontology Engineering”. *SIGMOD Record*, v. 31, n. 4, pp. 12–17.

STAAB, S., STUDER, R., SCHNURR, H., et al., 2001, "Knowledge Processes and Ontologies". *IEEE Intelligent Systems,* v. 16, n. 1, pp. 26–34.

STUDER, R., BENJAMINS, V. R., FENSEL, D., 1998, "Knowledge Engineering: Principles and Methods". *Data & Knowledge Engineering*, v. 25, n. 1-2, pp. 161–97.

SURE, Y., STAAB, S., STUDER, R., 2009, "Ontology Engineering Methodology", In: *Handbook on Ontologies*, pp. 135–52. *International Handbooks on Information Systems*, Springer Berlin Heidelberg.

TANG, S., HANNEGHAN , M., December 2011, "Game Content Model: An Ontology for Documenting Serious Game Design". In: *DESE '11 Proceedings of the 2011 Developments in E-systems Engineering*, pp. 431–36, Dubai, UAE.

TANG, S., HANNEGHAN , M., CARTER, C., 2013. "A Platform Independent Game Technology Model for Model Driven Serious Games Development". *Electronic Journal of E-Learning*, v. 11, n. 1, pp. 61–79.

TEKINBAS, K. S., ZIMMERMAN, E., 2003, *Rules of Play: Game Design Fundamentals*. Cambridge, Massachusetts, The MIT Press.

THORN, A., 2013, *Game Development Principles*. 1 ed, Boston, MA, Cengage Learning PTR.

USCHOLD, M., GRUNINGER, M., 1996, "Ontologies: Principles, Methods and Applications". *The Knowledge Engineering Review*, v. 11, n. 2, pp. 93–136.

USCHOLD, M., JASPER, R., 1999,"A Framework for Understanding and Classifying Ontology Applications". *IJCAI-99 Workshop on Ontologies and Problem-Solving Methods (KRR5)*, Stockholm, Sweden.

USCHOLD, M., KING, M., 1995, "Towards a Methodology for Building Ontologies". *Workshop on Basic Ontological Issues in Knowledge Sharing, Held in Conjunction with IJCAI-95*, Quebec, Canada.

USCHOLD, M., TATE, A., 1998, "Putting Ontologies to Use". *The Knowledge Engineering Review*, v. 13, n. 1, pp 1–3.

WALKER, J.. 2011, "Deus Ex: HR's Boss Fights Were Outsourced". Available at: <http://www.rockpapershotgun.com/2011/09/19/deus-ex-hrs-boss-fights-were-outsourced/>. Acessed: 2015-07-24 15:41:25.

WINGET, M. A., SAMPSON, W. W., 2011, "Game Development Documentation and Institutional Collection Development Policy", In: *Proceedings of the 11th Annual International ACM/IEEE Joint Conference on Digital Libraries*, pp. 29–38, *JCDL '11*, New York, NY, USA, ACM.

WONGTHONGTHAM, P., CHANG, E., DILLON, T., et al., 2009, "Development of a Software Engineering Ontology for Multisite Software Development". *IEEE Transactions on Knowledge and Data Engineering*, v. 21, n. 8, pp. 1205–1217.

ZAGAL, J. P., MATEAS, M., FERNÁNDEZ-VARA, C., et al., June 2005, "Towards an Ontological Language for Game Analysis". In: *DiGRA '05 - Proceedings of the 2005 DiGRA International Conference: Changing Views: Worlds in Play*, pp. 3–14, Vancouver, Canada.

Appendix A – OWL 2

In this appendix, I will introduce OWL and OWL 2 to readers not familiar with the language. All information presented in this appendix has been compiled from several sources listed on Section A.6. First, the intended goals and design of OWL will be presented. Second, the different versions of OWL will be detailed. Third, the most important features of OWL and OWL 2 are presented. Fourth, OWL limitations will be presented. Finally, the references used for this appendix are listed.

This appendix is a compilation of OWL and OWL 2 originated from the W3C (2004a, 2004b, 2012a, 2012b, 2012c) website e OWL tutorials (HORRIDGE, 2011, STEVENS *et al.*, 2013).

A.1 – What is OWL?

The OWL language is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF Schema (RDF-S) by providing additional vocabulary along with a formal semantics (W3C, 2004a). OWL is a revision of the DAML+OIL (W3C, 2001) web ontology language incorporating lessons learned from the design and application of DAML+OIL. OWL is part of the growing stack of W3C recommendations related to the Semantic Web:

- XML provides a surface syntax for structured documents, but imposes no semantic constraints on the meaning of these documents.
- XML Schema is a language for restricting the structure of XML documents and also extends XML with datatypes.
- RDF is a datamodel for objects ("resources") and relations between them, provides a simple semantics for this datamodel, and these datamodels can be represented in an XML syntax.
- RDF Schema is a vocabulary for describing properties and classes of RDF resources, with a semantics for generalization-hierarchies of such properties and classes.
- OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.

OWL is not a programming language but a declarative language because it describes a state of affairs in a logical way. Appropriate tools, like reasoners, can then be used to infer further information about that state of affairs. Also, it is not a schema language for syntax conformance. Unlike XML, OWL does not provide elaborate means to prescribe how a document should be structured syntactically.

OWL is not a database framework. Admittedly, OWL documents store information and so do databases. Moreover a certain analogy between assertional information and database content as well as terminological information and database schemata can be drawn.

However, usually there are crucial differences in the underlying assumptions. If some fact is not present in a database, it is usually considered false (closed-world assumption) whereas in the case of an OWL document it may simply be missing (but possibly true), following the open-world assumption.

A.2 – OWL Types

OWL has three increasingly-expressive sublanguages: OWL Lite, OWL DL, and OWL Full.

- OWL Lite supports those users primarily needing a classification hierarchy and simple constraints. For example, while it supports cardinality constraints, it only permits cardinality values of 0 or 1. It also has a lower formal complexity than OWL DL.
- OWL DL (BAADER *et al.*, 2007) supports those users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, while a class may be a subclass of many classes, a class cannot be an instance of another class). OWL DL is so named due to its correspondence with description logics (DL), a field of research that has studied the logics that form the formal foundation of OWL.
- OWL Full is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right. It is unlikely that any reasoning software will be able to support complete reasoning for every feature of OWL Full.

A.3 – OWL Features

The features presented are of the OWL DL sublanguage. OWL has several types of constructors. They are separated in distinct categories.

A.3.1 – Basic Elements

Most of the elements of an OWL ontology concern classes, properties, instances of classes, and relationships between these instances. The following features of this type are:

- Class: It defines a group of individuals that belong together because they share some properties. For example, *Falcon* and *Owl* are both members of the class *Bird*. Classes can be organized in a specialization hierarchy using subclasses.
- Subclass: Class hierarchies may be created by making one or more statements that a class is a subclass of another class. For example, the class *Guitar* is stated to be a subclass of the class *Music Instrument*. Thus, reasoners can deduce that if an individual is a *Guitar*, then it is also a *Music Instrument*.
- Property: They can be used to state relationships between individuals or from individuals to data values. Examples of properties include *hasWeapon* and *hasHealthPoints*. The first can be used to relate an instance of a class Warrior to an instance of the class *Weapon* (and are thus occurrences of *ObjectProperty*), and the

second can be used to relate an instance of the class Warrior to an instance of the datatype Integer (and is thus an occurrence of *DatatypeProperty*).

- Sub-property: Property hierarchies may be created by making one or more statements that a property is a sub-property of one or more other properties. For example, *hasSword* is stated to be a sub-property of *hasWeapon*. Thus, reasoners can deduce that if an individual is related to another by the *hasSword* property, then it is also related to the other by the *hasWeapon* property.
- Domain: A domain of a property limits the individuals to which the property can be applied. If a property relates an individual to another individual, and the property has a class as one of its domains, then the individual must belong to the class. For example, the property *hasWeapon* is stated to have the domain of Warrior. Thus, reasoners can deduce that if Guts *hasWeapon* Dragon Slayer, then Guts must be a *Warrior*.
- Range: The range of a property limits the individuals that the property may have as its value. If a property relates an individual to another individual, and the property has a class as its range, then the other individual must belong to the range class. For example, the property *hasWeapon* is stated to have the range of Weapon. Thus, reasoners can deduce that if Glenn is related to Masamune by the *hasWeapon* property, (i.e., Masamune is the weapon of Glenn) then Masamune is a *Weapon*.
- Individuals: They are instances of classes, and properties may be used to relate one individual to another. For example, an individual named Tigrex is described as an instance of the class *Monster* and the property *hasHabitat* relates the individual Tigrex to the individual Ancestral Steppe which is an instance of the class *Habitat*.

A.3.2 – Equality and Inequality

The following features are related to equality or inequality:

- Equivalent Class: Two classes may be stated to be equivalent. Equivalent classes have the same instances. Equality can be used to create synonymous classes. For example, *Ghost* is stated to be *equivalentClass* to *Phantom*. Thus, reasoners can deduce that any individual that is an instance of *Ghost* is also an instance of *Phantom* and vice versa.
- Equivalent Property: Two properties may be stated to be equivalent. Equivalent properties relate one individual to the same set of other individuals. Equality may be used to create synonymous properties. For example, *hasFriend* may be stated to be the *equivalentProperty* to *hasAlly*. Thus, reasoners can deduce that if X is related to Y by the property *hasFriend*, X is also related to Y by the property *hasAlly* and vice versa. A reasoner can also deduce that *hasFriend* is a *subProperty* of *hasAlly* and *hasAlly* is a *subProperty* of *hasFriend*.
- Same Individuals: Two individuals may be stated to be the same. These constructs may be used to create a number of different names that refer to the same individual. For example, the individual James Bond is stated to be the same individual as 007.
- Different Individuals: An individual may be stated to be different from other individuals. For example, the individual Mario may be stated to be different from the individual Luigi. Thus, if the individuals Mario and Luigi are both values for a property that is stated to be functional (thus the property has at most one value), then

there is a contradiction. Explicitly stating that individuals are different can be important in when using languages such as OWL (and RDF) that do not assume that individuals have one and only one name. For example, with no additional information, reasoners will not deduce that Mario and Luigi refer to distinct individuals.

A.3.3 – Property Characteristics

The following features are related to property characteristics:

- Object Property: Relations between instances of two classes. Discussed in Section A.2.1.
- Datatype Property: Relations between instances of classes and RDF literals or XML Schema datatypes. Discussed in Section A.2.1.
- Inverse: One property may be stated to be the inverse of another property. If the property P1 is stated to be the inverse of the property P2, then if X is related to Y by the P2 property, then Y is related to X by the P1 property. For example, if *controlsCharacter* is the inverse of *isControlledBy* and Player1 *controlsCharacter* Ryu, then reasoners can deduce that Ryu *isControlledBy* Player1.
- Transitive: Properties may be stated to be transitive. If a property is transitive, then if the pair (x,y) is an instance of the transitive property P, and the pair (y,z) is an instance of P, then the pair (x,z) is also an instance of P. For example, if *isBelow* is stated to be transitive, and if Block1 is below Block2 and Block2 is below Block3, then reasoners can deduce that Block1 is below Block3.
- Symmetric: Properties may be stated to be symmetric. If a property is symmetric, then if the pair (x,y) is an instance of the symmetric property P, then the pair (y,x) is also an instance of P. For example, *enemy* may be stated to be a symmetric property. Then a reasoner that is given that Mario is an *enemy* of Bowser can deduce that Bowser is an *enemy* of Mario.
- Functional: Properties may be stated to have a unique value. If a property is a functional, then it has no more than one value for each individual. The construct *functionalProperty* is shorthand for stating that the property's minimum cardinality is zero and its maximum cardinality is 1. For example, *hasKeyItem* may be stated to be functional. From this a reasoner may deduce that no individual of *Inventory* may have more than one key item. This does not imply that every *KeyItem* must be stored at an *Inventory*.
- Inverse Functional: Properties may be stated to be inverse functional. If a property is inverse functional then the inverse of the property is functional. Thus the inverse of the property has at most one value for each individual. This characteristic has also been referred to as an unambiguous property.

A.3.4 – Property Restrictions

OWL restrictions fall into three main categories (HORRIDGE, 2011):

- Quantifier Restrictions;
- Cardinality Restrictions;

- *hasValue* Restrictions.

Also, a restriction describes an anonymous class (an unnamed class). The anonymous class contains all of the individuals that satisfy the restriction, i.e., all of the individuals that have the relationships required to be a member of the class.

A.3.4.1 – Quantifier Restrictions

Quantifier restrictions can be further categorized into existential restrictions and universal restrictions.

- Existential restrictions describe classes of individuals that participate in at least one relationship along a specified property to individuals that are members of a specified class. Its OWL construct is *someValuesFrom*. In Protégé the construct is some.
- Universal restrictions describe classes of individuals that for a given property only have relationships along this property to individuals that are members of a specified class. Its OWL construct is *allValuesFrom*. In Protégé the construct is only.

As seen before, domain and range restrict the types of the elements that make up a property. These mechanisms are global as they apply to all instances of the property. Existential and universal restrictions are local to their containing class definition.

The *allValuesFrom* restriction requires that for every instance of the class that has instances of the specified property, the values of the property are all members of the class indicated by the *allValuesFrom* clause. For example, the weapon of an *Archer* must be a *Bow*. The *allValuesFrom* restriction is on the *hasWeapon* property of this *Archer* class only. *Swordsman* that use *Sword* as a *Weapon* are not constrained by this local restriction.

The *someValuesFrom* restriction is similar. If *allValuesFrom* is replaced with *someValuesFrom* in the example above, it would mean that at least one of the *hasWeapon* properties of an *Archer* must point to an individual that is a *Bow*. The difference between the two formulations is as follows:

- allValuesFrom: For all archers, if they have weapons, all weapons are bows.
- someValuesFrom: For all archers, they have at least one weapon that is a bow.

The first does not require an archer to have a weapon. If it does have one or more, they must all be bows. The second requires that there be at least one weapon that is a bow, but there may be weapons that are not bows.

A.3.4.2 – Cardinality Restricitions

The OWL construct cardinality permits the specification of exactly the number of elements in a relation. For example, we specify *Car* to be a class with exactly four *Wheel* instances.

The construct *maxCardinality* can be used to specify an upper bound. The construct *minCardinality* can be used to specify a lower bound. In combination, the two can be used to limit the property's cardinality to a numeric interval.

A.3.4.3 – Value Restrictions

The OWL construct *hasValue* allows us to specify classes based on the existence of particular property values. Hence, an individual will be a member of such a class whenever at least one of its property values is equal to the *hasValue* resource.

For example, all *Dragon* instances are of the fire element. That is, their *hasElement* property must have at least one value that is equal to *Fire*. As for *allValuesFrom* and *someValuesFrom*, this is a local restriction. It holds for *hasElement* as applied to Dragon.

A.3.5 – Complex Classes

OWL provides additional constructors with which to form classes. These constructors can be used to create so-called class expressions. OWL supports the basic set operations, namely union, intersection and complement. These are named *unionOf*, *intersectionOf*, and *complementOf*, respectively. Additionally, classes can be enumerated. And it is possible to assert that class extensions must be disjoint. Complement classes will not be described as they are not supported by Protégé.

A.3.5.1 – Intersection Classes

An intersection class is described by combining two or more classes using the *intersectionOf* construct which is the same as a logical AND operator. For example, the intersection of *Human* and *Male* describes an anonymous class that contains the individuals that are members of both classes, it also means that it is a subclass of both classes. The anonymous intersection class above can be used in another class description. For example, *Man* is a subclass of the anonymous class described by the intersection of *Human* and *Male*. In other words, *Man* is a subclass of *Human* and *Male*.

A.3.5.2 – Union Classes

A union class is created by combining two or more classes using the *unionOf* construct which is the same as a logical OR operator. For example, the union of *Man* and *Woman* describes an anonymous class that contains the individuals that belong to either the class *Man* or the class *Woman* (or both). The anonymous class that is described can be used in another class description. For example, the class *Person* might be equivalent of the union of *Man* and *Woman*.

A.3.5.3 – Enumerated Classes

As well as describing classes through named superclasses and anonymous superclasses such as restrictions, OWL allows classes to be defined by precisely listing the individuals that are the members of the class. For example, the class *DaysOfTheWeek* contains the individuals (and only the individuals) Sunday, Monday, Tuesday, Wednesday, Thursday, Friday and Saturday. Classes such as this are known as enumerated classes.

A.3.5.4 – Disjoint Classes

In principle, OWL does not prevent classes to 'overlap'. Therefore it cannot be assumed that an individual is not a member of a particular class simply because it has not been asserted to be a member of that class. In order to separate a group of classes, they must be made disjoint from one another using the *disjointWith* construct. This ensures that an individual who has been asserted to be a member of one of the classes in the group cannot be a member of any other classes in that group. For example, *Male* and *Female* are made disjoint from each other. This means that an individual that is member of the *Male* class cannot be member of the *Female* class and vice-versa.

A.4 – OWL 2 New Features

OWL 2 is an extension and revision of the OWL language developed by the W3C Web Ontology Working Group and published in 2004 (referred to hereafter as "OWL 1"). Like OWL 1, OWL 2 is designed to facilitate ontology development and sharing via the Web, with the ultimate goal of making Web content more accessible to machine. OWL 2 adds new functionality with respect to OWL 1. Some of the new features are syntactic sugar while others offer new expressivity, including:

- keys;
- property chains;
- richer datatypes, data ranges;
- qualified cardinality restrictions;
- asymmetric, reflexive, and disjoint properties.

A.4.1 – Property Chains

OWL 1 does not provide a means to define properties as a composition of other properties, as uncle could be defined; hence, it is not possible to propagate a property (*locatedIn*) along another property (*partOf*). The OWL 2 construct *ObjectPropertyChain* in a *SubObjectPropertyOf* axiom that allows a property to be defined as the composition of several properties. It allows the definition of relationships among three individuals; the most prominent example is the property uncle which may be defined as chain of parent and brother properties.

A property may be chained even with itself. For example, we may define property *isEmployedAt*, which is a chain of itself and the transitive property *isPartOf*, meaning that if a person is employed at some organizational unit, the person is also employed at the bigger organizational units.

A.5 – OWL 1 and OWL 2 Limitations

OWL has some limitations that should be considered before it is used.

A.5.1 – Difference between Classes and Individuals

There are important issues regarding the distinction between a class and an individual in OWL. A class is simply a name and collection of properties that describe a set of individuals. Individuals are the members of those sets. Thus classes should correspond to naturally occurring sets of things in a domain of discourse, and individuals should correspond to actual entities that can be grouped into these classes. In building ontologies, this distinction is frequently blurred in two ways:

- Levels of representation: In certain contexts something that is obviously a class can itself be considered an instance of something else. For example, *Fire Dragon* is an example instance of the class *Dragon*, as it can denote an actual fire dragon. However, *Fire Dragon* could itself be considered a class, the set of all actual fire dragons.
- Subclass vs. instance: It is very easy to confuse the instance-of relationship with the subclass relationship. For example, it may seem arbitrary to choose to make *Fire Dragon* an individual that is an instance of *Dragon*, as opposed to a subclass of it. This is not an arbitrary decision. The *Dragon* class denotes the set of all dragons, and therefore any subclass of *Dragon* should denote a subset of dragons. Thus, *Fire Dragon* should be considered an instance of Dragon, and not a subclass. It does not describe a subset of dragons, it is a dragon.

A.5.2 – Expressivity Limits

KUBA (2012) explains that OWL 1 cannot express the uncle relation, which is a chain of relations parent and sibling. OWL 2 can express uncle using property chains, however it still cannot express relations between individuals referenced by properties. For example, OWL 2 cannot express the child of married parents concept because it cannot express the relationship between parents of the individual. Because OWL 2 DL is a fragment of first order predicate logic, it cannot express the following:

- Fuzzy expressions - “It often rains in autumn.”
- Non-monotonicity - “Birds fly, penguin is a bird, but penguin does not fly.”
- Propositional attitudes - “Eve thinks that 2 is not a prime number.” (It is true that she thinks it, but what she thinks is not true.)
- Modal logic (CHELLAS, 1980)
  - Possibility and necessity - “It is possible that it will rain today.”
  - Epistemic modalities - “Eve knows that 2 is a prime number.”
  - Temporal logic - “I am always hungry.”
  - Deontic logic - “You must do this.”

Bibliographic References

BAADER, F., CALVANESE, D., MCGUINNESS, D. L., et al., 2010, *The Description Logic Handbook: Theory, Implementation and Applications*. 2 ed, New York, NY, USA, Cambridge University Press.

CHELLAS, B. 1980, *Modal Logic: An Introduction*, Cambridge University Press.

HORRIDGE, M., 2011, *Protégé OWL Tutorial*. Available at: <http://owl.cs.manchester.ac.uk/publications/talks-and-tutorials/protg-owl-tutorial/>. Accessed: 2015-09-18 23:37:42.

KUBA, M., 2012, *OWL 2 and SWRL Tutorial*. Available at: <http://dior.ics.muni.cz/~makub/owl/>. Accessed: 2015-09-20 23:16:24

STEVENS, R., STEVENS, M., MATENTZOGLU, N., et al., 2013, *Manchester Family History Advanced OWL Tutorial.* Available at: <http://owl.cs.manchester.ac.uk/publications/talks-and-tutorials/fhkbtutorial/>. Accessed: 2015-09-18 23:38:06

W3C, 2001, *DAML+OIL Reference Description*. Available at: <http://www.w3.org/TR/daml+oil-reference>. Accessed: 2015-11-01 18:26:59

———, 2004a, *OWL Web Ontology Language Guide*. Available at: <http://www.w3.org/TR/owl-guide/>. Accessed: 2015-09-20 22:49:26

———, 2004b, *OWL Web Ontology Language Overview*. Available at: <http://www.w3.org/TR/2004/REC-owl-features-20040210>. Accessed: 2015-09-18 17:17:13

———, 2012a, *OWL 2 Web Ontology Language Document Overview (Second Edition)*. Available at: <http://www.w3.org/TR/2012/REC-owl2-overview-20121211/>. Accessed: 2015-09-21 00:02:36

———, 2012b, *OWL 2 Web Ontology Language New Features and Rationale (Second Edition)*. Available at: <http://www.w3.org/TR/owl2-new-features/>. Accessed: 2015-09-20 23:59:10

———, 2012c, *OWL 2 Web Ontology Language Primer (Second Edition)*. Available at: <http://www.w3.org/TR/owl2-primer/>. Accessed: 2015-09-21 00:03:14

Appendix B – Conceptualization Tables

In this appendix, the tables produced during the conceptualization phase and modified during the implementation and evaluation phases of the ontology development process are presented. As it was presented in Chapter 7, there are four types of tables: concepts, attributes, relations and axioms. Furthermore, it is recommended that Appendix A is read before if the reader has no knowledge of OWL 2. Those tables are going to be used as a base and reference for the implementation of the ontology in OWL 2. Gray colored rows represent classes, relations and axioms that were not implemented.

For each module, the following tables will be presented: taxonomy (hierarchy of sub-classes), relations and axioms. Note that some modules may not have some of the tables. There will be only one data property table that will contain all data properties of the ontology. The taxonomy table columns are:

- Name: Self-explanatory. Concepts cannot have the same name;
- Parent: Indicates that the concept is a specialization of another concept;
- Disjoint with: Indicates the other concepts that the concept is disjoint with. The concept can only be disjoint with concepts that have the same parent;
- Level: The depth of the concept in the ontology taxonomy.

The data properties table columns are:

- Name: Self-explanatory;
- Concept: The concept that has the attribute;
- Datatype: Indicates if the attribute is a string or integer for example;
- Description: A brief description of the attribute;
- Constraints: Describe any constraints that the attribute may have.

The relations table columns are:

- Name: Self-explanatory;
- Domain: Indicates the concepts that are the beginning of the relation;
- Range: Indicates the concepts that are the end of the relation;
- Card. (Cardinality): Indicates the number of instances of the property a concept can have;
- Properties: Indicates whether the relation is symmetrical, functional, etc.;
- Inverse: Indicates if the relation has an inverse relation.

The axioms table columns are:

- Code: A unique identifier for the axiom;
- Axiom: A description of what the axiom is.

Table 7 – Game Object module taxonomy

Name	Parent	Disjoint with	Level
Game Object	-	Attribute, Event, Action, State, EO	1
Space	Game Object	Space Connection	2
0D Space	Space	1D, 2D, 3D Space	3
1D Space	Space	0D, 2D, 3D Space	3
2D Space	Space	0D, 1D, 3D Space	3
3D Space	Space	0D, 1D, 2D Space	3
Discrete Space	Space	Continuous Space	3
Continuous Space	Space	Discrete Space	3
Bounded Space	Space	Unbounded Space	3
Unbounded Space	Space	Bounded Space	3
Display Object	Game Object	-	2
Display Space	Bounded Space, Display Object	-	4
Player Object	Game Object	-	2
Player View	Display Space	-	5

Table 8 – Game Object module relations

Name	Domain	Range	Card.	Property	Inverse
hasGOPart	Game Object	Game Object	0..N	Transitive	isPartOfGO
isPartOfGO	Game Object	Game Object	0..1	Transitive	hasGOPart
containsGO	Game Object	Game Object	0..N	Transitive	isContainedInGO
isContainedInGO	Game Object	Game Object	0..1	Transitive	containsGO
handlesEvent	Game Object	Event	1..N	-	isHandledBy
sendsGOEvent	Game Object	Internal Event	0..N	-	isSentByGO
hasGOState	Game Object	State	1..N	-	isStateOfGO
performsAction	Game Object	Action	1..N	-	isPerformedBy
hasGOAttribute	Game Object	Attribute	1..N	-	isAttributeOfGO
sendsOutput	Game Object	Input	1..N	-	outputSentByGO
receivesInput	Game Object	Output	1..N	-	inputReceivedByGO
hasAsset	Game Object	Asset	0..N	-	assetOfGO
isGOofVG	Game Object	Video Game	0..N	-	hasGO

Table 9 – Game Object module axioms

#	Axiom
1	A GO that contains more than two instances of another GO **implies** that it has a Collection of the contained GO

Table 10 – Attribute module taxonomy

Name	Parent	Disjoint with	Level
Attribute	-	GO, Event, Action, State, EO	1
Simple Attribute	Attribute	Collection, File	2
Spatial Attribute	Simple Attribute	-	3
Position	Spatial Attribute	-	4
Atomic Attribute	Simple Attribute	Composite Attribute	3
Composite Attribute	Simple Attribute	Atomic Attribute	3
Collection	Attribute	File, Simple Attribute	2
File	Attribute	Collection, Simple Attribute	2
Number	Atomic Attribute	String, Boolean, Enumeration	4
String	Atomic Attribute	Number, Boolean, Enumeration	4
Boolean	Atomic Attribute	Number, String, Enumeration	4
Enumeration	Atomic Attribute	Number, String, Boolean	4
Time	Number	Time Flow	5
Time Flow	Number	Time	5
Discrete Time	Time	Continuous Time	6
Continuous Time	Time	Discrete Time	6
Output Attribute	Attribute	-	2
Visual Attribute	Output Attribute	-	3

Table 11 – Attribute Module relations

Name	Domain	Range	Card.	Property	Inverse
compositePart	Composite Att.	Composite Part	2..N	-	partOfComposite
partOfComposite	Composite Part	Composite Att.	0..1	Functional	compositePart
collectionOf	Collection	GO, Attribute	1	Functional	-
isChangedByAction	Attribute	Action	0..N	-	changesAttribute
canBeChangedByAction	Attribute	Action	0..N	-	canChangeAttribute
determinesState	Attribute	State	1..N	-	isDeterminedBy
determinesEvent	Attribute	Event	1..N	-	isTriggeredByAttribute
attributeOfGO	Attribute	GO	0..1	Functional	hasGOAttribute
attributeOfEvent	Attribute	Event	0..1	Functional	hasEventAttribute
attributeOfEO	Attribute	EO	0..1	Functional	
isParameterOf	Attribute	Action	0..1	Functional	hasParameter

Table 12 – Attribute module axioms

#	Axiom
2	An Attribute can only be changed by Actions that belong to the same Game Object
2a	A Composite Attribute can be only composed by Atomic or Composite Attributes
3	A Attribute triggers an Event if it determines the State that triggers the Event

Table 13 – Event module taxonomy

Name	Parent	Disjoint with	Level
Event	-	GO, Attribute, Action, State, EO	1
Internal Event	Event	External Event	2
External Event	Event	Internal Event	2
Spatial Event	Event	-	2
Timed Event	Event	-	2

Table 14 – Event module relations

Name	Domain	Range	Card.	Property	Inverse
isHandledBy	Event	Game Object	0..N	-	handlesEvent
hasEventAttribute	Event	Attribute	0..N	-	isAttributeOfEvent
isTriggeredByState	Event	State	1..N	-	triggersEvent
isTriggeredByAttribute	Event	Attribute	0..N	-	determinesEvent
causesAction	Event	Action	1..N	-	isCausedByEvent
isSentByGO	Internal Event	Game Object	1..N	-	sendsGOEvent

Table 15 – Event module axioms

#	Axiom
4	An External Event is an Event triggered by an External State
5	An Internal Event is an Event triggered by an GO State

Table 16 – Action module taxonomy

Name	Parent	Disjoint with	Level
Action	-	GO, Attribute, Event, State	1
Spatial Action	Action	-	2
Timed Action	Action	-	2
Player Action	Action	-	2
Output Action	Action	-	2

Table 17 – Action module relations

Name	Domain	Range	Card.	Property	Inverse
changesAttribute	Action	Attribute	0..N	-	isChangedByAction
canChangeAttribute	Action	Attribute	0..N	-	canBeChangedByAction
hasCondition	Action	State	1..N	-	isConditionOf
hasParameter	Action	Attribute	0..N	-	isParameterOf
nextAction	Action	Action	0..N	-	previousAction
previousAction	Action	Action	0..N	-	nextAction
hasOutcome	Action	State	0..N	-	isOutcomeOf
isCausedByEvent	Action	Event	0..N	-	causesAction
isPerformedBy	Action	Game Object	1	Functional	performsAction
hasActionState	Action	State	1	Functional	isStateOfAction
affectsOutput	Action	Output	1..N	-	outputAffectedByAction

Table 18 – Action module axioms

#	Axiom
6	An Action has a Previous Action if the Action condition is the same as the Previous Action outcome
7	An Action has a Next Action if the Action outcome is the same as the Next Action condition
8	It is impossible that the condition State is the same as the outcome State

Table 19 – State module taxonomy

Name	Parent	Disjoint with	Level
State	-	GO, Attribute, Event, Action, EO	1
GO State	State	External State	2
External State	State	GO State	2
Spatial State	State	-	2
Timed State	State	-	2

Table 20 – State module relations

Name	Domain	Range	Card.	Property	Inverse
hasStatePart	State	State	0..N	Transitive	isPartOfState
isPartOfState	State	State	0..N	Transitive	hasStatePart
nextState	State	State	1..N		previousState
previousState	State	State	0..N		nextState
equivalentState	State	State	0..N	Symmetric and Transitive	-
isDeterminedBy	State	Attribute	1..N		determines
triggersEvent	State	Event	1..N		isTriggeredBy
isStateOfGO	State	Game Object	1	Functional	hasGOState
isStateOfEO	State	EO			hasEOState
isConditionOf	State	Action	1..N	-	hasCondition
isOutcomeOf	State	Action	1..N	-	hasOutcome
isPartOfCondition	State	Action	1..N	-	-
isPartOfOutcome	State	Action	1..N	-	-
isPartOfOutput	State	Output	1..N	-	-
isStateOfAction	State	Action			hasActionState
outputsAsset	State	Asset	0..N	-	outputtedIn

Table 21 – State module axioms

#	Axiom
9	A State that is part of a condition State is also a condition State
10	A State that is part of an outcome State is also an outcome State
11	If the State that the Attribute determines is part of another State then the Attribute also determines it
12	A GO State is a State of a Game Object instance
13	An External State is a State of an External Object instance
14	If a State is a condition State of an Action and the Action changes the Attribute that determines the State then its Next State is the Action outcome State (or part of the outcome State)
15	If a State is an outcome State of an Action and the Action changes the Attribute that determines the State then its Previous State is the Action condition State (or part of the condition State)
15a	A Divisible State is an State that has at least 2 parts (States).

Table 22 – Space module relations

Name	Domain	Range	Card.	Property	Inverse
isConnectedTo	Space	Space	0..N	Symmetric	-

Table 23 – Space module axioms

#	Axiom
16	A Discrete Space is composed of 0D Spaces
17	A 0D Space must be connected to another 0D Space
17a	Spaces have Spatial Attributes such as position, shape or speed.
18	Spatial States are States determined by Spatial Attributes
19	Spatial Actions change Spatial Attributes
20	Spatial Events are triggered by Spatial States

Table 24 – Time module relations

Name	Domain	Range	Card.	Property	Inverse
hasEquivalence	Time	Time	0..N	Symmetric	-
isFlowOf	Time Flow	Continuous Time	1	Functional	hasFlow
hasFlow	Continuous Time	Time Flow	1..N	-	isFlowOf

Table 25 – Time module axioms

#	Axiom
21	A Timed State is determined by a Time Attribute and it has a duration data property
22	A Timed Action must have a Timed State
23	A Timed Event is triggered by a Timed State

Table 26 – External Object module taxonomy

Name	Parent	Disjoint with	Level
External Object	-	GO, Attribute, Event, Action	1
Hardware	External Object	Software, Player	2
Software	External Object	Hardware, Player	2
Video Game	Software	-	3
Player	External Object	Hardware, Software	2

Table 27 – External Object module relations

Name	Domain	Range	Card.	Property	Inverse
hasEOPart	EO	EO	0..N	Transitive	isPartOfEO
isPartOfEO	EO	EO	0..1	Transitive	hasEOPart
hasEOAttribute	EO	Attribute	1..N	Inverse Functional	isAttributeOfEO
hasEOState	EO	State	1..N	-	isStateOfEO
sendsInput	EO	Input	1..N	-	inputSentByEO
receivesOutput	EO	Output	1..N	-	outputReceivedByEO

Table 28 – Hardware module relations

Name	Domain	Range	Card.	Property	Inverse
hasHWPart	Hardware	Hardware	0..N	Transitive	isPartOfHW
isPartOfHW	Hardware	Hardware	0..1	Transitive	hasHWPart
runSW	Hardware	Software	0..N	-	isRunByHW
manipulatedBy	Hardware	Player	1..N	-	interactsWith

Table 29 – Hardware module axioms

#	Axiom
24	Hardware only sends Physical Input
25	Hardware only receives Physical Output

Table 30 – Software module relations

Name	Domain	Range	Card.	Property	Inverse
inRunByHW	Software	Hardware	1	Functional	runSW
hasModule	Software	Software	0..N	Transitive	isModuleOf
isModuleOf	Software	Software	0..N	Transitive	hasModule

Table 31 – Software module axioms

#	Axiom
26	Software only sends Non-physical Input
27	Software only receives Non-physical Output

Table 32 – Player module relations

Name	Domain	Range	Card.	Property	Inverse
interactsWith	Player	Hardware	1..N	-	manipulatedBy
controlsGO	Player	Player Object	1..N	-	controlledBy
controlledBy	Player Object	Player	1..N	-	controlsGO
playerSendsInput	Player	Player Input	1..N	-	inputSentByPlayer
inputSentByPlayer	Player Input	Player	1..N	-	playerSendsInput
playerReceivesOutput	Player	Player Input	1..N	-	outputReceivedByPlayer
outputReceivedByPlayer	Player Input	Player	1..N	-	playerReceivesOutput

Table 33 – Player module axioms

#	Axiom
28	A Player sends Input and receives Output through interaction with a Hardware
29	Physical Input is a Player Input if it is sent from the Hardware that the Player interacts with
30	Game Object is a Player Object if it handles Player Input
31	Player Action is an Action caused by a Player Input
32	Physical Output is a Player Output if it is received from the Hardware that the Player interacts with
33	Player View is a Display Space that sends Player Output

Table 34 – Input module taxonomy

Name	Parent	Disjoint with	Level
Input	External Event	-	3
Physical Input	Input	Non-physical Input	4
Image Input	Physical Input	Touch Input, Sound Input	5
Sound Input	Physical Input	Image Input, Touch Input	5
Touch Input	Physical Input	Image Input, Sound Input	5
Player Input	Physical Input	-	5
Non-physical Input	Input	Physical Input	4

Table 35 – Input module relations

Name	Domain	Range	Card.	Property	Inverse
inputSentByEO	Input	EO	1..N	-	sendsInput
inputReceivedByGO	Input	Game Object	1..N	-	receivesInput

Table 36 – Input module axioms

#	Axiom
34	A Non-physical Input can only be sent by a Software
35	A Physical Input can only be sent by a Hardware

Table 37 – Output module taxonomy

Name	Parent	Disjoint with	Level
Output	State	-	2
Physical Output	Output	Non-physical Output	4
Player Output	Physical Output	-	5
Video Output	Physical Output	-	5
Audio Output	Physical Output	-	5
Mechanical Output	Physical Output	-	5
Non-physical Output	Input	Physical Output	4

Table 38 – Output module relations

Name	Domain	Range	Card.	Property	Inverse
outputSentByGO	Output	Game Object	1..N	-	sendsOutput
outputReceivedByEO	Output	EO	1..N	-	receivesOutput
outputAffectedByAction	Output	Action	1..N	-	affectsOutput

Table 39 – Output module axioms

#	Axiom
36	An Output cannot be a condition (or part of one) of an Action
37	An Output Action is an Action that has an Output as an outcome
38	An Output Attribute is an Attribute that is an Parameter of an Output Action
39	An Action affects an Output if it changes an Output Attribute
40	A Game Object sends an Output if it has an Output Action
41	A Non-physical Output can only be received by a Software
42	A Physical Output can only be received by a Hardware

Table 40 – Video Output module axioms

#	Axiom
43	A Video Output Action is an Action that has a Video Output as an outcome
44	A Display Space is a Space that sends Video Output
45	A Display Object is a Game Object that sends Video Output
46	A Visual Attribute is an Attribute that is an Parameter of a Video Output Action

Table 41 – Asset module taxonomy

Name	Parent	Disjoint with	Level
Asset	-	-	1
Visual Asset	Asset	-	2
Textual Asset	Asset	Visual Asset, Audio Asset	2
Audio Asset	Asset	-	2

Table 42 – Asset module relations

Name	Domain	Range	Card.	Property	Inverse
assetOfGO	Asset	Game Object	1..N	-	hasAsset
fileType	Asset	File	1	Functional	-
outputtedIn	Asset	State	1..N	-	outputsAsset

Table 43 – Asset module axioms

#	Axiom
47	If a State outputs an Asset it implies that the State is a condition of an Output Action

Table 44 – Video Game module relations

Name	Domain	Range	Card.	Property	Inverse
hasGO	Video Game	Game Object	1..N	-	isGOofVG

Table 45 – VGDO data properties

Name	Class	Datatype	Description
name	Any	String	Self-explanatory
description	Any	String	Self-explanatory
datatype	Atomic Attribute	Any	Datatype of the attribute
initialValue	Atomic Attribute	Any	Initial value
minValue	Number	Any Number	Minimum value
maxValue	Number	Any Number	Maximum value
duration	Timed State	Integer	Duration of the State
developer	Software	String	Name of the developer
genre	Video Game	String	Genre of the video game

Appendix C – Competency Questions

In this appendix, I present the competency questions that are used to evaluate the VGDO. The questions are separated in two categories: internal and external modules. The *External Object* module does not have competency questions because the *Software*, *Player* and *Hardware* modules are natural extensions of it.

C.1 – Internal Module Questions

- What are the actions of a game object?
- What are the possible states of a game object?
- What are the events that the game object handles?
- What are the attributes of a game object?
- Which objects have an attribute of a certain type?
- What are the actions that change the value of a certain attribute?
- Which events are triggered by the value of a certain attribute?
- What are the actions that an event can possibly call?
- What are the attributes of an event?
- What are the events triggered by a certain state?
- What are the game objects that receive a certain event?
- What are the parameters of an action?
- What are the attributes that an action changes?
- What are the conditions for an action to be performed?
- What are the possible next states that of a certain state?
- What are the possible previous states of a certain state?
- What are the possible events that are triggered by a certain state?
- Which game objects are contained in a space?
- Which game objects have bounded spaces?
- Is a certain space connected to other spaces?
- What is the duration of an action?
- What is the duration of a state?
- What are the events triggered by time?
- What are the time attributes of the game?

C.2 – External Module Questions

- What physical inputs does a certain hardware send?
- What physical outputs do a certain hardware receives?
- Which software runs in the hardware besides the game?
- What non-physical inputs does a certain software send?
- What non-physical outputs do a certain hardware receives?
- What game objects does the player control?
- What inputs are available for the player to make?
- What are the outputs that the player receives?
- What game objects react to player inputs?
- What game object actions does an input invoke?
- What are the types of input present in the game?
- What are the outputs of a video game?
- What actions change a certain output?
- What events call an action that changes output?
- Which game objects have assets?
- What are the visual assets of the game?
- What are the audio assets of the game?
- What are the textual assets of the game?
- What is the name of the video game?
- What is the hardware used to play the video game?
- What is the software that communicates with the video game?
- What is the genre of the video game?