A Domain-Specific Language for Computer Games

Master Thesis

PUBLIC VERSION

Jeroen Dobbe

A Domain-Specific Language for Computer Games

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Jeroen Dobbe
born in Voorburg, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Cannibal Game Studios
Belvédèrebos 85
2715 VD Zoetermeer
the Netherlands
`www.cannibalgamestudios.com`

A Domain-Specific Language for Computer Games

Author: Jeroen Dobbe
Student id: 1149741
Email: j.dobbe@cannibalgamestudios.com

Abstract

The development of computer games is currently done using ad hoc development methods and middleware and tools that are not particularly well suited for supporting every member of a game development team in their day to day work. Some members of the development team have established helpful development tools, while others are lacking any support whatsoever.

This thesis is aimed at finding current bottlenecks and the paradigm (way of thinking) used when it comes to specifying games and to apply domain-specific languages, where possible, to further advance the current toolset that is being used in the games industry. This domain-specific language (DSL) will, of course, be aimed mostly at the part of a game that needs the most programming: game design (e.g. objects, rules and interaction).

During this master thesis project I have done research in the area of a domain-specific language for computer games, defined such a DSL and implemented it using the Cannibal Engine as a supporting class library. The domain-specific language contains elements to specify objects, states, interactions, game-flow, rules and storylines. Based on this domain-specific language, game studios will (in the future) be able to create games faster, centered around a well-defined game design.

Preface

After having finished most of my academic programme it was time for me to start my thesis work. Of course, having started a company in the computer games industry, and working on it pretty much full-time, did not leave a lot of room to just stop and work on my thesis for a year. I had to find something that would allow me to combine the two.

Having discussed this with several people, I decided on the assignment to somehow combine *domain-specific languages* with *computer games*, to allow the games industry to benefit from the advantages of domain specific languages. This thesis describes the result of that assignment: the search for the right application of domain-specific languages, the definition of the language, a description of the IDE, the relevant underlying support framework, a full working example and an evaluation of the language.

Acknowledgments

First and foremost I would like to thank my colleagues for supporting me and putting up with me and my ramblings during the course of the past few years. Special thanks goes to Remco for endless discussions and for always questioning everything I come up with and to Jerke for constantly keeping me grounded by reminding me that things have to be practical in the real-world.

I would also like to thank the Software Engineering Research Group group at TU Delft for allowing me to do this thesis and build my company at the same time. Special thanks here goes to Leon Moonen for helping me find an interesting thesis subject that not only helped me complete my academic programme, but also moved the company forward. Of course I would like to thank Eelco Visser, my supervisor, for providing me with valuable advice, guidance and insight.

A word of thanks also goes out to the different members of the Benelux Games Initiative for providing me with valuable insight into the game development community in the Benelux region and their game development processes.

Two professors in particular have been very valuable throughout my study. Stijn Oomes; for making me *very* aware of the user and teaching me a mind-set for usability that will last a lifetime. Rafael Bidarra; for always believing in the Cannibal

initiative and supporting us in everything Cannibal Game Studios has done so far. I had the pleasure to assist them both in running and building student curriculum projects.

My parents also deserve honorable mentioning, for putting me through university and always supporting me 100% with all my different endeavours over the past few years. Especially my father, who has patiently read through this thesis several times.

And last, but certainly not least, my girlfriend Liselotte, for the support and for her understanding with me working late hours, weekends and sometimes through the night.

Jeroen Dobbe

Delft, the Netherlands
July 25th, 2007

Contents

Chapter 1

Introduction

Game developers usually work for 2-3 years on one game title. Because of this, there are big financial risks involved in developing a major game. More and more, game developers are starting to use middleware gaming engines to reduce the time-to-market of their games. These engines are aimed at streamlining the development process and helping the game developers by providing resources for rapid prototyping and rapid iteration.

Cannibal Game Studios [49] is a company that develops innovating middleware and tools for the games industry, both for entertainment and serious purposes (e.g. simulation, training). One of the main products developed at Cannibal Game Studios is a gaming engine. Strong selling points for this Cannibal Engine are its capability to allow for rapid and iterative development through an easy to use class library based on managed code. This class library provides a level of abstraction that reduces the implementation details when it comes to hardware handling.

Despite the current middleware efforts, the development of a computer game still takes an enormous amount of time. To support the development of the current 'next-gen' games, Cannibal Game Studios feels that it will be important to establish *next-gen* game development practices and methodologies. One way to improve the productivity, the maintainability and the overall quality of game software would be to implement a *domain-specific language* for computer games [33, 56].

Unlike general purpose programming languages, domain-specific languages are not aimed at providing a language general enough to be used across a wide variety of problem domains. A domain-specific language (DSL) is a programming language geared towards solving problems in a specific domain. Examples of domain-specific languages include Fran [15] and Stratego [58].

Currently, conventional object-oriented language are used for the development of computer games, but these are not particularly suited for the development of games. They usually contain no support for states and timing and they require the developer to think about a lot of details concerning, for example, data management, data representation and rendering. It would be better if we could abstract away from these details and create a language which does not confront the developers with these details while programming the game itself. This will allow game developers to concentrate more on the look & feel of the game, instead of hard-to-debug implementation details that slow down development profoundly.

The goal of this master thesis project is to do research into the area of a domain-specific language for computer games, define such a DSL and implement it using Cannibal as a supporting class library.

In this thesis I will present my research into the areas of domain-specific languages (chapter 2), related work (chapter 3) and games and their developers (chapter 4). I will disuss the Cannibal Game Development Platform in the context of this project and the DSL in chapter 5. Several existing languages and their paradigms are discussed in chapter 6.

The requirements and considerations that go into the design of the DSL are summarized in chapter 7. I will conclude with an evaluation of the designed language (chapter 9) and the conclusions that can be drawn from this thesis project (chapter 10).

Chapter 2

Domain-Specific Languages

According to [57] a *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. In other words: a domain-specific language is a (programming) language designed with a specific purpose in mind. Using such a DSL, applications for a specific domain may be developed with a better focus on the problem domain. This may lead to increases in productivity, reliability and maintainability, all of which should have a positive impact on the quality of the product and the time-to-market. Which the production-team behind *Duke Nukem Forever* [1] could probably use.

I start in section 2.1 with a discussion of the software engineering context in which domain-specific languages are used. In section 2.2 the advantages and disadvantages of developing and using domain-specific languages are discussed. Section 2.3 discusses the different methodologies one could follow when developing a DSL, in section 2.4 the methodology used for analyzing the computer games domain is discussed. I conclude in section 2.5 with a short discussion of the different tools available for developing a domain-specific language.

2.1 Context

Domain-specific languages are usually used in a broader context of software engineering. In software engineering, model-driven engineering [32] or generative programming [12] are both methodologies in which (part of) the source code for the products is generated from representations at a higher, domain-specific abstraction level. In model-driven engineering, models are used to represent the implementation at some abstraction level. These models are used as an integral part of the software development process. Generative programming uses similar techniques in which code is generated from higher level (code) representations.

According to this way of thinking domain-specific languages can be considered as a modeling language for defining software at a higher level of abstraction. However, usually the distinction is made between a model and a language. Where a model is some graphical representation of the software, for instance a UML class diagram [18], which is used to specify the structure of the software. A (domain-specific) language is considered as the means to implement the logic of an application, or the 'real' implementation. Specifying and displaying software in either way can be seen as giving shape to (or modeling) the software.

Domain-specific languages can be used by developers to specify the software, or parts of the software, in a more domain related fashion than is normally the case. Using domain concepts and terminology it is possible to even have domain experts specify the software, instead of 'real' programmers. These programmers can design the DSL and the tools used to make them work. Grammars are used to specify the language and tools are generated for parsing, transforming and compiling the resulting software.

Recently the TU Delft started a research project [13] in which the need for evolutions of models and code (and domain-specific languages) is recognized. Since it will be hard to capture all necessary requirements and details up-front and considering the rapid changes game development is under, it is important to make sure the DSL provides a solid basis for game development, while remaining extendible and customizable where necessary.

2.2 Advantages and Disadvantages

Before we set out and talk about the different aspects of creating a computer games DSL, we will first stop and consider the advantages and disadvantages that come with using a DSL in any domain. The developers at Cannibal Game Studios feel that the advantages outweigh the disadvantages for this project and are therefore optimistic about the creation of the computer games DSL.

One of the most important reasons for using a DSL in this domain is that the overhead of having to deal with technical details when implementing the gameplay is minimized as much as possible by providing a language that allows programmers, or event game designers, to work at the right abstraction level. Another advantage of working at the correct abstraction level is that design information is retained and the code is more or less self-documenting, at least more than code in a general purpose language. Knowledge and terminology of the domain is used to represent the program and this makes it easier to understand and better suited for reuse.

Working at a higher abstraction level and a better understandable and documented source code will inevitably lead to better maintainability, higher productivity and better reliability of the end-products. Besides that, software written using a DSL is usually more portable to different platforms than software written in a general purpose language containing more technical (and platform specific) details.

Of course there are also a number of disadvantages of using a DSL. Designing, implementing and maintaining a DSL is a rather elaborate job and this involves a lot of costs. Since the DSL is created for the use in a middleware product, this can be seen as no major disadvantage since the costs may be amortized over many different end-products. During the development of the middleware some difficult choices have to be made to make the DSL successful: there should be a balance between the domainspecificity and generality of the language and the language must be properly scoped to prevent it from being either too small or too large.

Besides the costs and difficulties of the development of a DSL, the users also must be taught how to work with the DSL, this may also lead to costs. First the users must be trained to understand the language and know the different concepts and constructs, then the users will have to start using the language to get experienced and get to the same, or higher, productivity level.

Using a DSL may lead to a loss in efficiency of the final software, but it may also lead to an increase. When using a DSL you write code at a higher abstraction level, translating this code into a general purpose language or some other target language may lead to inefficiencies in the final product. This can be due to code sections that are translated one-on-one with function calls on a library. Using hand written code to represent both can potentially execute faster. However, when you write code at a higher abstraction level, the intentions of the programmer are more clear. This, combined with the knowledge of the domain may lead to more efficient compilation, leading to faster code.

For the computer games DSL efficiency is of great importance; games should run in real-time. It is important to take the efficiency of the DSL compilation into consideration when developing the DSL. A small loss in efficiency might be allowed to gain a bigger increase in productivity.

2.3 Development of a DSL

Developing a domain-specific language is an elaborate process, which contains a number of important steps. [57] gives a number of important steps to be completed, [15] adds to this by emphasizing the importance of the environment tools for a DSL. Combined they give the following steps:

1. Define the problem domain and gain knowledge about the semantic notions and commonly used operations.
2. Design the DSL that describes applications in this problem domain.
3. Construct a support library that supports the operations and semantic notions.
4. Implement the domain specific language (e.g. using a compiler).
5. Provide the environment tools needed to develop programs in the DSL (e.g. debuggers and editors).
6. Provide examples, manuals and rewrite programs using the DSL.

For the purpose of this master project I will mainly be concentrating on steps 1, 2 and 4, 5 and 6. Step 3 has, for the most part, already been taken care of by the Cannibal Engine. However, during this thesis work some interesting findings let to adaptation and restructuring of the engine, as to provide support for the DSL. Steps 5 and 6 are present as proof-of-concepts, and they should most likely be further developed and integrated into Cannibal Game Studios' strategy when using the DSL in conjunction with their engine.

To perform the domain analysis I will take a look at the different methodologies available to do domain analysis. In section 2.4 I will give a short overview of different domain analysis methodologies in use. The different tools supporting steps 4 and 5 are discussed in section 2.5. Relevant details about the underlying support library, the Cannibal Game Development Platform, are contained in chapter 5.

2.4 Domain Analysis

For the analysis of a problem domain several (slightly) different methods have been developed. In this section I will take a look at three of these approaches and determine what must be done in order to analyze the computer games domain. Most of the information here is taken from [12].

2.4.1 Feature-Oriented Domain Analysis

Feature-Oriented Domain Analysis (FODA) is one of the most mature and best documented domain analysis methods currently available. It is based on features and introduces feature models and feature modeling. Features are, in the FODA method, end-user-visible characteristics of a system.

The process behind FODA consists of the following two phases:

1. Context Analysis: Defining the boundaries and scope of the domain to be analyzed.

2. Domain Modeling: Producing a *domain model* and the *domain dictionary*.

The domain model consists of an *information model*, which could be represented by an entity-relationship diagram, semantic network or some object-oriented diagrams. Another important element is the *feature model*, which documents the different commonalities and differences among related systems within the domain. The *operational model*, representing the behavioral relations between the objects in the information model and the features in the feature model, is also a part of the domain model.

The domain dictionary consists of a list of the terminology used in the domain and can be used as a reference for finding keywords and constructs for the DSL.

2.4.2 Organization Domain Modeling

Organization Domain Model (ODM) [46] is a second mature and well documented domain analysis method. The ODM process consists of three steps:

1. Plan Domain: Determine stakeholders and their objectives, scope the domain and define the domain consisely.

2. Model Domain: Acquire domain information, describe the domain and create the domain model.

3. Engineer Asset Base: Produce the architecture for the systems in the domain, quite like step 3 described in section 2.3.

These three steps are further subdivided twice to reach a level of workable tasks, some of these tasks will be used in our method and are described in more detail in section 2.4.4. Step one and two are typical domain analysis steps, while step three is more aimed at implementation of some 'solutions'.

ODM contains a number of different aspects that are worth mentioning in relation to this project:

- Focus on stakeholders and context, this constitutes a user-centered approach, which is one of the approaches emphasized by Cannibal Game Studios.
- Binding site, which allows for fine-grained classification of the features on the specific binding times: compile-time, start, runtime, debug-time, etc.
- ODM uses a tailorable process, the domain modeling and methods used to analyze the domain, this allows for flexibility in the process of retrieving information.

2.4.3 Domain-Specific Software Architecture

The Domain-Specific Software Architecture (DSSA) approach was developed with an emphasis on the central role of the software architecture concept. This approach also consists of three main steps that lead to three products:

1. Domain model: A model of the domain, more rudimentary than the ones discussed above.

2. Reference Requirements: These are the features described above, where each feature is either a functional or a non-functional requirement.

3. Reference Architecture: An architecture for a family of systems within the domain, consisting of an architecture model, feature model, design record (description of the components) and constrains and rationale.

The DSSA approach uses Architecture Description Language to formally describe the architecture and the interrelationships between its components. This shows the emphasis put on describing the (reference) software architecture for the specific domain.

2.4.4 Analyzing the Computer Games Domain

To arrive at an overview of the domain the DSL will cover we must take a number of steps. The method described here will lead us to define the domain more clearly than has been done in the introduction and will give specific boundaries for the domain. It will also provide an overview of domain terminology and the different commonalities and variations in the computer games domain. The product of the method used will lead to a mental model of the domain which will be used for designing, specifying and implementing the DSL.

Most of the methods described in section 2.3 give at least two major results of the domain analysis:

- Domain definition; A concise definition of the problem domain including the boundaries and restrictions.
- Domain model; A (feature) model of the domain containing the domain terminology, concepts and relations between concepts.

The details of how to arrive at these two results, and what these results actually contain are different for each method. During this domain analysis I have chosen to include a number of steps and a specific view on the end results.

Steps to Take

I must first get acquainted with the domain and see what the status quo is in the field of DSLs for computer games (see chapter 3). This allows me not only to get some basis from which we can look further into the specifics of the domain, but it also gives pointers to best practices and pitfalls that are typical to the computer games domain.

After a check of the status quo it is important to know whether these insights and products are actually used in practice. Examining current practice also gives insight into the workings and mindset/paradigm used at game studios. Looking at the commonalities and differences in the development process allow us to see what features are actually of concern to game studios. Not only are domain experts interviewed, but I also take a look at the games that have been developed and at some aspects of game source code (see chapter 4).

After having studied the domain we will focus attention on the Cannibal Game Development Platform: the platform of which the DSL will be an important part (see chapter 5). This game development platform will allow developers to work at higher levels of abstraction, while retaining their expressive power at lower levels. Looking at the technology behind and composition of this development platform will give insight into the specific requirements and methods used.

The Results

Using the information gained from following the steps above, we will be able to give the two results discussed above (the domain definition and the domain model). The domain definition will be given by specifying the focus point of the DSL based on the needs of game developers, what is learned from related work and what the role of the DSL will be in the Cannibal Game Development Platform.

The domain model will be given by providing a diagram which contains the most important concepts of the domain and their relations. This, together with the API of the Cannibal Engine [50] will provide a basis for the operations and constructs in the DSL.

2.5 Tools

Before I start with the second part of the research; a treatment and analysis of the problem domain, I will first briefly consider the different tools available to a developer of a domain-specific language. The tool to be used in the implementation phase should, of course, be particularly suited to this specific project and the choice of the tool is therefore an important one.

According to [30] the use of a 'Language Prototyping Tool' (LPT) can greatly improve the reliability, repairability, portability and documentation of a language. Using a good tool or language for implementing the DSL also proves to be more time and cost effective. These features come from the fact that a LPT provides better code locality and a better structuring for the language syntax and semantics which also provides self-documentation.

2.5.1 The ASF+SDF Meta-Environment

The ASF+SDF (Algebraic Specification Formalism + Syntax Definition Formalism) Meta-Environment is an interactive development environment for the automatic generation of interactive systems for constructing language definitions and generating tools for them [55]. The Meta-Environment is meant to support the user (developer of a DSL) in creating and implementing a domain-specific language. It allows for the creation of support tools like a parser, pretty-printer, editor, debugger and compiler.

The ASF+SDF Meta-Environment is based on the ASF+SDF formalism, which allows the developer to specify both the syntactic and the semantic aspects of the domain-specific language. Based on these definitions the meta-environment can generate the tools necessary to support development in the new domain-specific language.

2.5.2 Stratego/XT

Stratego/XT [58, 59] is the combination of the Stratego language for strategic programming and the XT bundle of transformation tools. The Stratego language allows for specifying program transformations over ATerms [54] using rewrite rules. It has many features that are quite interesting, and allow for very flexible and powerful translations, like generic traversals and dynamic rules.

The XT bundle of transformation tools provides many tools that are needed for building a program transformation system. With these tools Stratego/XT becomes a complete package that can be used to develop the computer games DSL from start to end. The programs that are written using Stratego/XT are compiled to C code and can thus also be compiled to Windows, which is our target platform. The only thing we need to worry about is the fact that not all Stratego libraries will work with Windows.

Stratego/XT is rather similar to ASF+SDF, but it has some advantages. Most notably the availability of expertise in the form of Martin Bravenboer and Eelco Visser, two developers of Stratego/XT, the large user base and the transformation language Stratego.

2.5.3 Intentional Programming

Intentional Programming or Intentional Software [45], as it is now called, is a quite different approach from the two toolsets described above. With Intentional Programming the aim is to capture the *intention* of the programmer and use this paradigm to write programs. In practice this comes down to constructing a programming in two different ways. On the one hand the programmer writes a program, on the other hand the intentions of the programmer are also created as extensions to the programming environment and language.

This is done by separating source code from its representations. The source of a program is referred to as *active source* which is basically an (annotated) abstract syntax tree with a number of methods defined over it for presentation, compilation, debugging, etc. This provides a great deal of flexibility over the language in which programs are written. The language can be extended by adding new intention which complement not only the language, but also the editing, presentation, compilation, etc.

Having the source represented directly by abstract syntax tree has some advantages: it saves parsing, makes reasoning about the source code easier and helps during refactoring (references are used to refer to definitions). But it also has a disadvantage: source is saved in some binary format, which means you require an Intentional Programming IDE to work with it.

Extending (or creating) languages with Intentional Programming is relatively easy. You implement intentions by creating a library interface and then implementing reduction, rendering, debugging, editor and other methods to extend the IP environment and the programming language. Some extensions APIs are available to work with reduction, rendering, etc. Compiling the code is done through reducing the tree(s) to more simple and lower-level tree(s).

Intentional Programming was developed by Charles Simonyi who used to work at Microsoft. Charles Simonyi stopped work on IP at Microsoft and is now developing Intentional Software at his own company [11], while Microsoft is focusing its attention towards the Microsoft DSL Tools [36]. Currently Intentional Software, is not in a state in which it can be used, but it might be interesting to start using this when it becomes available for maintaining and using this DSL. It might even be worthwhile to look at combining the Stratego program translation language with Intentional Software, as this might be a very powerful combination.

2.5.4 Microsoft DSL Tools

Microsoft DSL Tools is a suite, nested inside Visual Studio, that allows users to specify domain specific languages, build a graphical designer and define code generators. This DSL tools is a important part of the the Software Factories initiative envisioned by Microsoft [21].

Using Microsoft DSL Tools, users can develop a graphical language inside Visual Studio which is used to generate code based on templates. This code can then be compiled and run like any other application written using a .NET language.

2.5.5 Manual Approach

Besides using an existing tool it is always possible to do things by hand from the start. This usually provides more flexibility and better control over details, however, it is a lot more work. One of the reasons one might have for using the manual approach is the lack of availability of a tool that integrates well with existing technology. Greater control over exact implementation and workings may also be desired.

Chapter 3

Related work

This chapter will give a short overview of the work that has already been done in the area of domain specific languages for computer games. I start out with an example of a widely used gaming engine which has a scripting language to program the game itself in. This Unreal engine is described in section 3.1. Another widely used 'engine' is Game Maker, software written by Mark Overmars to enable the easy creation of 2d sprite based games. We will take a look at Game Maker in section 3.2.

In sections 3.3 and 3.4 some examples of domain specific languages for small subsets of the gaming domain are given. Since these languages are used for a smaller application domain, these languages are most likely of a higher abstraction level than can be attained for a general game DSL, but they can still prove useful as an example.

In section 3.5 I describe different industry file formats as they may give insight into different storage formats and the terminology, notation and approaches used for those formats. They may also prove useful if they provide storage for certain types of behavior in any form.

3.1 Unreal

The technology behind the Unreal game series is called the Unreal engine. This Unreal engine is created and maintained by Epic Games. This technology is actually not just an engine, it also contains many tools and editors that can be used to help the game developer create the entire game. It also has a workflow associated with it in the form of a content pipeline.

Since the Unreal Engine is comparable to the Cannibal Game Development Platform (CGDP) when it comes to features and scope, it might be interesting to take a closer look at the Unreal Engine. For example, the purpose of UnrealScript is to make programming games easier by creating a language more geared towards game development. UnrealKismet is a visual scripting language aimed at level designers. Since the design goals for these two products are quite similar to those of CGDP, examples and ideas from these two products can be very useful for this project.

We start in section 3.1.1 by briefly explaining the framework behind the Unreal engine and the process of creating a game with the unreal engine. In section 3.1.2 we will take a further look at UnrealScript, the scripting language that allows programming with the unreal engine. We will also take a look at UnrealKismet, the visual scripting language, in section 3.1.3. We conclude in section 3.1.4 with some interesting remarks made by Tim Sweeney about programming games and the future.

3.1.1 Framework

The framework that makes up the Unreal Engine consists not only of the core engine technology, but it also contains content creation tools for creating levels, particle effects and animations. It also has a support infrastructure that provides the necessary support for game developers when they are creating their game.

The main philosophy behind the Unreal Engine is that artists should be able to create content for the game in a visual environment with as little programmer assistance as possible. Programmers are given a modular and extensible software library for building the game.

UnrealEd is the main (and most important) editor that comes with Unreal. The main goal of UnrealEd is to connect everything together. Programmers write and define objects and behavior in scripts, artists create the assets that are needed in a level: sound and graphics. The level editor can, using these assets and scripts, create a world in which the player will play the game.

3.1.2 UnrealScript

Programming with the Unreal engine is done through a scripting language called UnrealScript. This UnrealScript is designed to abstract from details like pointers and to provide a language to express concepts needed for game programming. In this way UnrealScript can be considered a domain-specific language geared towards the development of games. UnrealScript remains, however, at a fairly low programming level, which raises worries as to the abstraction level that can be attained for a game DSL.

According to [52] UnrealScript had three major design goals:

- To support the major concepts of time, state, properties and networking, which are lacking from traditional programming languages;
- To provide Java-style programming simplicity, object-orientation and compile-time error checking;
- To enable rich, high level programming in terms of game objects and interaction.

UnrealScript basically is a general purpose language with added functionality to accomplish the goals above. To achieve a java-style (rich and high-level) programming language in terms of objects and interaction, UnrealScript is made to be object oriented. Besides that, UnrealSript provides easy access to API calls and access to other objects to define the interaction. By compiling the script to some proprietary format, error checking can be done during compilation. Objects can expose properties to the outside world that may be used by level designers or game designers to customize the objects. These are all things you would expect to see from normal object oriented languages.

One of the major additions of UnrealScript to general-purpose programming language is the concept of states. Classes may contain definitions of methods, variables and states. In terms of object orientation, states can be seen as special subclasses, which define certain specific types of behavior (methods) and variables. States can add or overwrite variables and methods that are in the 'root' of the class. States can even inherit from each other.

```
// Base Attacking state.
state Attacking
{
// Stick base functions here...
}

// Attacking up-close.
state MeleeAttacking expands Attacking
{
// Stick specialized functions here...
}

// Attacking from a distance.
state RangeAttacking expands Attacking
{
// Stick specialized functions here...
}
```

Besides states, support for time-based programming is also provided through the use of latent functions. Latent functions are functions that do not return immediately after they have executed, but after a certain amount of time, or at a certain moment. Examples of such functions include:

Sleep(float Seconds) pauses the execution for a certain amount of time. This function returns after the given amount of seconds.

FinishAnim() pauses execution until the current animation sequence has completed. The time this function takes depends on the current animation and how long it needs to finish. This can be used to script based on the timing of animations.

FinishInterpolation() pauses execution until the current interpolation movement has completed. Again the time it takes depends on the interpolation. This function can also be used to script based on timing of animations/interpolations.

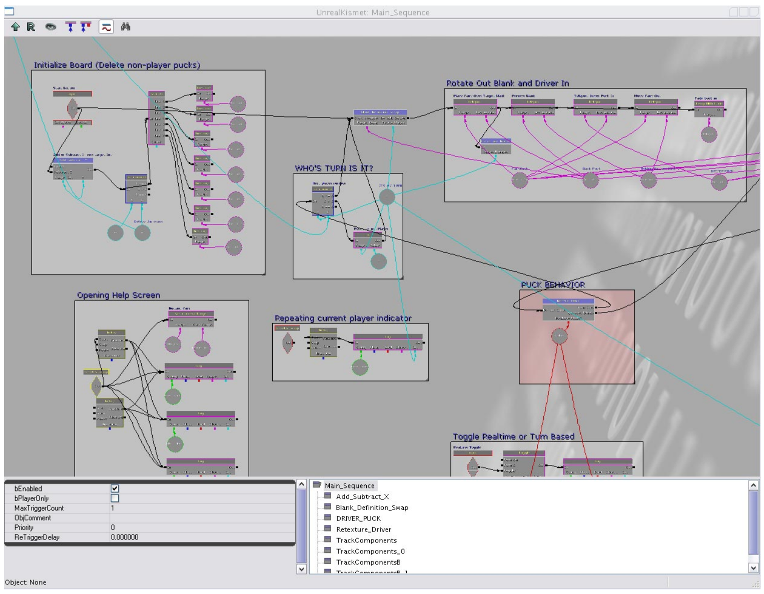
Using these latent functions it becomes possible for the programmer to base the execution of certain sections of code on the timing of other game elements. This is an often needed functionality in game programming and is often done by using standard trickery. Abstracting away from this trickery will prevent a lot of errors that can be made when applying the tricks. When many timing issues are combined the code can become complicated to read and understand, while the concepts underlying the code are relatively simple.

While UnrealScript provides support for different game related programming issues, it is still a low-level, general-purpose language. It just contains a few extra features that make it more suited for game development. When we take a closer look at UnrealKismet we will see that some elements can be abstracted to a higher level, which is more in the direction of the DSL.

3.1.3 UnrealKismet

In the latest version of the engine, Unreal Engine 3.0, Epic has also included UnrealKismet (or VisualKismet), a visual scripting language, which is supposed to be at a higher abstraction level and suited for use by game and level designers directly. Its main purpose is to provide the designer with a system to design levels without help from the programmer. The designer can do this by scheduling game events in a flow-chart-like format. You can use game events and specific actions and tie this all together using more general purpose combinatory elements.

The language behind Kismet is an event driven language (see section 6.2), which basically means that you tell the system what to do in case an event happens (e.g. explode when you get hit). These actions can be glued together so they happen conditionaly, delayed, in sequence, etc.



bEnabled	☑
bPlayerOnly	☐
MaxTriggerCount	1
ObjComment	
Priority	0
ReTriggerDelay	0.000000

Object: None

Figure 3.1: UnrealKismet

Below I will explain the basic elements that make up UnrealKismet, we will first deal with Events and Actions. After that I discuss Variables followed by some combinatory operators in the form of gates, conditionals and delays. I will also discuss hierarchical subdivision which are accomplished using sequences. Connections between all of these elements can be made by creating connections from outputs of elements to inputs of other elements. ’Impulse’ travels through these connections to activate elements. Impulse can be thought of as electrical current travelling through wires.

Events & Actions

Two of the most elementary elements of UnrealKismet are events and actions. Events are generate by objects or interactions between objects. For example touching a button object in the world can generate an event which can be attached to a button effects action. If you connect this button effects action to a toggle action, which you then connect to a light, toggling the light on and off, you have effectively created a light switch. This situation is depicted in figure 3.2.

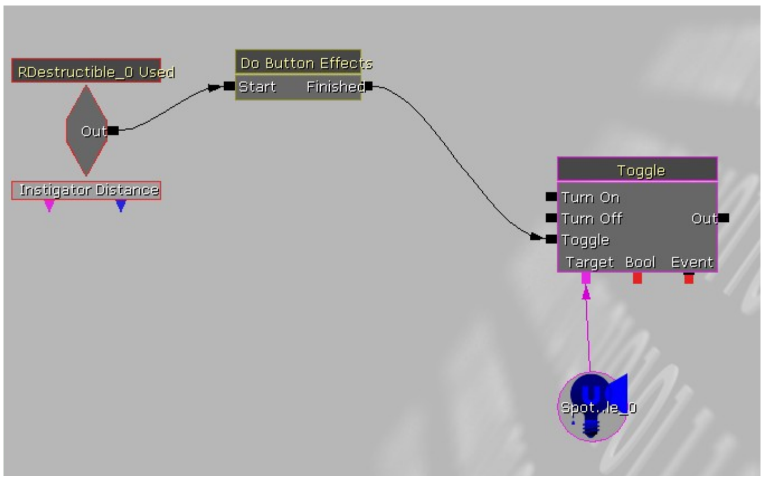


Figure 3.2: Connecting events and actions

Variables

Variables, like in any other language, can be used to either store references to objects dynamically. Meaning they will be assigned based on the events or actions and their compositions. They can also be used to store references to objects in a static way. These do not change throughout execution, but can still be used to connect to actions.

Combinations

With UnrealKismet you can also combine the different impulses to do some special actions. *Gates* can be used to either allow an impulse to pass or not to pass, depending on how it is set. It has an ’open’, ’close’ and ’toggle’ input to set whether the gate allows the impulse to pass. It also has an ’in’ and an ’out’ which indicate the input and output respectively.

A conventional ’if’ statement can be created using *conditionals*; these provide a way of checking for certain conditions. An example of a conditional is a ’compare objects’, where two objects are checked to be equals. Using variables you can input the objects and depending on the outcome you can allow impulse to travel elsewhere.

Delays are quite straightforward: they allow impulse to be delayed for a set amount of time. This amount of time can be ’hardcoded’ by setting it at programming time, but it can also be retrieved from some variable or object.

Sequences

Since a visual scripting language runs a reasonable risk of getting into trouble by getting too cluttered, UnrealKismet allows for hierarchically composing the visual script by using sequences. Sequences are a set of elements connected in some way, which can then be used as one element for further composition. Sequences can be given their own inputs and outputs to parameterize these new elements.

3.1.4 Tim Sweeney

Tim Sweeney from Epic Games recently gave a talk [53] about the programming language of the future from a game developers perspective. During this talk he spoke about his vision of a future programming language and the problems with conventional programming languages when creating games. First three kinds of code used in games are identified:

- Gameplay simulation
- Numeric computation
- Shaders

Four problem areas are then identified which might be improved upon by future languages to better suit the needs of (game) developers. Here I discuss the four problems identified and the relation to the game DSL. I will not go into the specific details of each problem area, but I will sketch the things to take into account when specifying the language in the next project phase.

Performance

One of the main important things in real-time (interactive) applications is the performance. You need to update many objects, while still retaining processing speeds that allow for 60 frames to be displayed every second. This means that every piece of game code that runs in real-time should execute as fast as possible. However, productivity for games is considered to be even more important. Many developers, Epic Games included, would be glad to sacrifice a bit of performance for an increase in productivity.

A thing that must be considered when looking at games is that there are no obvious bottlenecks, everything is interconnected and there are several non-trivial ways of improving performance. Considering the fact that game simulation takes up only about 10% of total CPU time might show that other areas are the first to tackle when trying to improve performance.

Modularity

One of the most important functionalities when writing any software libraries is modularity. It must be possible to easily extend, customize and reuse parts of the library as well as reuse your own add-ons. Consider the following engine setup of actors and players in games:

```
package Engine;

class Actor
{
int Health;

}
class Player extends Actor
{

}
```

If we want to extend this for the purpose of one game, adding extra functionality to the Actor, we can do this. But the Engine.Player does not inherit any information from the extended actor: GearsOfWar.Actor. We actually would like something where the Player used by the GearsOfWar package would also contain this functionality. This makes it a lot more easy and clear how to extend parts of the class library and really do some effective coding. Below is a code segment illustrating this:

```
Package GearsOfWar extends Engine;

class Actor extends Engine.Actor
{
// Here we can add new members
// to the base class.
}
class Player extends Engine.Player
{
// Thus virtually inherits from
// GearsOfWar.Actor
}
```

Reliability

Some things that go wrong with current programming language concerning dynamic values and using these include:

- Array out-of-bounds errors
- Null pointers
- Integer overflow
- Uninitialized variables

About half of the programming errors come from these problems. It will be important to keep these common problems in mind when defining the game DSL to make it more reliable and less error-prone. Making a lot of mistakes and having a hard time debugging these mistakes will have a significant impact on the productivity.

Concurrency

With the direction hardware is moving in, it might be interesting to look at concurrency. Ways of improving the performance of code by executing things in parallel. Tim Sweeney has some interesting remarks about how to accomplish concurrency for the three different types of code. The one most applicable to this project will be the concurrency theories for the gameplay simulation, we come back to this in section 6.3.

3.2 Game Maker

Game Maker [39] is an Integrated Development Environment (IDE) for creating (2d) computer games. The aim of Game Maker is to provide a completely integrated environment for construction the game. All elements of a game should come together: graphics, sound, music, gameplay, etc. It is created to be simple to use, with drag-and-drop-like features and capabilities. However, considerable power is retained through the use of a built-in scripting language. Game Maker is used mostly in education [40].

Game Maker is not focused on a particular kind of game, this means in practice that more work will have to be done to create a game than with software like Zillions Of Games (see section 3.4.2). But it does contain (basic) support for many features like sprite-based graphics, sound, music, user interface, configuration, in-game help, etc.

3.2.1 The Language

A game created with Game Maker consists of three different types of elements. First there is the game data containing sprites, sounds, backgrounds and fonts. These elements are also what is referred to by others as *assets*. The second element of a game is the control: objects, timelines, scripts and paths (animations). The third element consists of the levels or, as they are called in Game Maker, rooms. They basically define how the level (initially) looks and what objects and triggers are present.

Game Maker basically uses an event-driven model of programming (this model will be explained in more details in section 6.2). Objects in the game have a certain representation (sprites) and have a certain type of behavior. This behavior is defined using scripts, these scripts define actions to be taken when certain events happen. Events can be things like collisions, creations, timers, user input, etc.

Another paradigm used by Game Maker is the paradigm of Object-Orientation. Every object (class) in Game Maker can inherit behavior and attributes from parent objects and instances (object) of these objects represent real game objects. The user can create objects in a very easy way dragging-and-dropping events and scripts and actions together to define the behavior. Sprites and sound or music can be imported to represent the object. Of course all of the attributes can also be changed at run-time by using scripts.

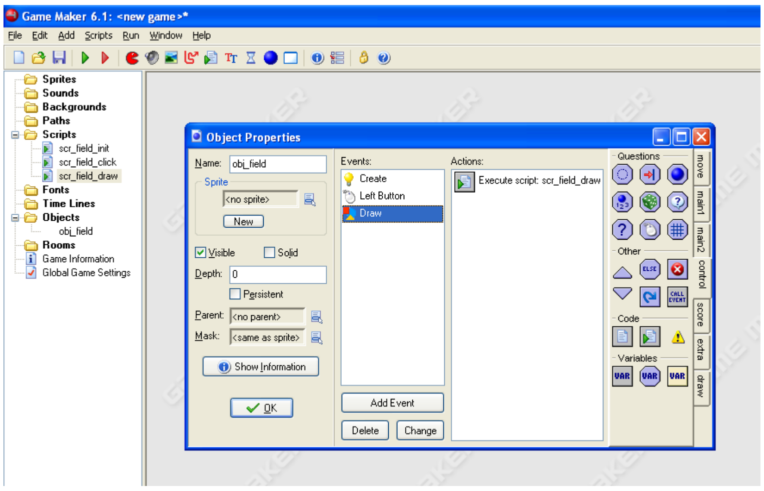
3.2.2 An example: Tic-Tac-Toe

When we want to implement a game like *Tic Tac Toe* using Game Maker this can be done in multiple ways. Here we briefly describe how the author of Game Maker has done it himself in one of his books [41].

The author created one *object*, which are the entities that are present in your game. This object is called *obj_field*. Once this object is created, behavior is added by writing the following scripts:

- *scr_field_init*: which initializes the playing field by creating a 3x3 matrix used to store the contents of the 9 squares (empty, X or O);
- *scr_field_click*: to handle the user clicking on one of the squares, and acting accordingly.
- *scr_field_draw*: to draw the playing field and communicate to the user the current state of the game and board.

Then we have to setup the *Create*, *Mouse Click* and *Draw* events to use these scripts when they fire. The result of this is shown in figure 3.3



Name: obj_field
Sprite: <no sprite>
[✓ Visible] Solid
Depth: 0
Persistent
Parent: <no parent>
Mask: <same as sprite>
Events: Create, Left Button, Draw
Actions: Execute script: scr_field_draw

Figure 3.3: Game Maker

Now we have a functional playing field, but some game logic still needs to be added to check whether one of the players has won, or when a draw has happened. For this we have to add four (!) scripts, which do some checking:

- *scr_check_player_win*: checks whether the player has won;
- *scr_check_computer_win*: checks whether the computer or second player has won;
- *scr_check_draw*: checks whether there is a draw;

- *scr_check_end*: combined the three scripts above and checks whether we are done playing for this round.

The *scr_check_end* should be called after every move to ensure that the correct end-situation is detected and acted upon.

To really finish this game one would have to add graphics (and sound), which can be done by adding sprites to the game and using the script *scr_field_draw* to draw these at the appropriate location depending on the current state of the board.

An important thing to realize here is that by actually implementing all these scripts, you still work at a very low-level, in a general purpose programming language: Game Maker Language (GML). This is the language you use to implement all the scripts. The actions game maker provide that you can use without writing scripts are also quite low-level, basically wrapping the built-in functions you can call from GML (e.g. *draw_sprite()* or *play_sound*).

So, while Game Maker is quite suited to prototype or build very simple games on your own, it is not very much suited for larger projects. Using the events and actions provided by Game Maker will not be enough to implement all the functionality you need for a bigger game. Even while implementing Tic Tac Toe, you will have to resort to scripting very early in the development process, and after a while Game Maker might even hamper development.

3.3 ViGL

ViGL stands for *Vi*deo *G*ame *L*anguage [29]. ViGL is a domain- specific language which attempts to capture commonality between 2d video games. This allows a game developer to do rapid prototyping and iterations for simple 2d video games. The code written in the DSL can be used to generate a codebase for the game.

While this project focuses on a smaller domain than this project, ViGL might still be a useful example of what can be attained using domain-specific languages in computer games. Therefore details about ViGL are gathered here and the useful aspects that might be used are summarized.

3.3.1 Design and Considerations

The ViGL language is chosen to be based on XML since it is very easy to understand, and the technology was in place to easily parse XML. Later on the authors realized that using XML was not ideal, they gave reasons like: it misses control-flow features and has a bloated syntax. This eventually led to a mix between XML (declarative) and some embedded code (imperative).

The authors of the ViGL language performed a domain analysis for 2d games. They give an overview of the features and concepts common to general 2d games. This overview is given in table 3.1.

All of these concepts translate well to a three-dimensional environment. Table 3.1 thus gives a basic overview of elements that are also present in the type of games the

Features	*Requirements*
Graphics	Loading of images, texts, etc.
Sound	Load, Play, Pauze and Stop music.
User Input	Keyboard and mouse input handling.
Objects	The objects of the game (different shapes, sizes and positions).
World	The game world and the rules and shape of the world.
Interaction	Interaction between objects in the game world.
User Control	The way the user controls the objects in the world.

Table 3.1: Overview of the 2d games domain

DSL should be designed for. Since many of these features can be considered as low-level details, the concepts here can be separated into two different layers of abstraction. On the game design level one would like to think in terms of the game world, game objects, the user, the interaction between objects and the rules of the game. More low-level are the specific details of graphics, sound, user input handling and other related details. The specific way in which these details are handled does not influence the game itself, only the representation of the game. These details *are* very important for the game - most of the time they even determine to a large extent the look & feel of the game - but they do not change the game dynamics and the design of the game.

This point makes it clear that besides the high level game design definitions, low-level details of graphics, sound, etc. should also be controlled in some way. This might be done by separating the two rigorously and connecting them through some mechanism. Another way might be to completely integrate both. As it is important to prevent unnecessary dependencies between team members, it might be wise to try and separate the two, so game designers and interaction designers are not troubled with details that artists and musicians might want to be confronted with.

3.3.2 Language Definition

If we look at the language definition of ViGL, we see that only part of the above mentioned features and requirements are implemented in the final language definition. However, there are some key points that we can extract from this language definition that is not clear from the description of ViGL. One of the main things that the language contains is the paradigm shift from thinking in classes and methods (standard OOP) to thinking in game objects, rules and a world, and most importantly, events and actions in that world that define behavior of items. This is a very useful concept which we will build upon in chapter 5.

At the moment the language definition of ViGL allows us to define some general game settings (such as resolution and full screen). It allows some basic code insertions at some points to allow for better extensibility, but this is not very flexible. Code is inserted as 'native' code: Ruby. This is the target language for the ViGL compiler. This does not separate the DSL from the target language, which means that it loses some of the advantages of DSLs (see section 2.2).

Apart from that, the language defines objects and possible events and ties these to a world with concrete instantiations of these objects, which might trigger these events.

Behavior of these objects is then defined by handlers of the triggered events. For the objects to have a graphical representation, the language also allows for specifying this on an object-to-object basis.

The things that are still missing from the language, which the authors admit to, are the actual game play aspects. Many low-level detail implementations were included, but due to time pressure, the game play aspects were not implemented. Things like player types, game types, world types, views and mechanisms for scoring and levels are all missing. These are typically things at a higher abstraction level and these should be aimed for in the game DSL.

3.3.3 An example: Tic Tac Toe

For a better understanding of ViGL it might be interesting to see an example of ViGL in action. Here we explain how a game like *Tic Tac Toe* would be implemented using ViGL. The complete source code for this example can be found in appendix B.

We start out with the ViGL tag, which indicates the start (and end) of the game. We can define resolution and full screen settings here. We choose only to define the resolution here, the default full screen setting (false) is fine for this purpose. The main things to implement here are the squares, which should react to a mouse click from the user. When the user clicks on the squares, they should become a specific color (or depict a cross or circle) to mark the square as belonging to one of the players. We therefore create an object definition (objectdef) which defines the default look for the square and a method for handling a mouse click event.

Handling the mouse click event involves checking whether the mouse is inside the square, the mouse down event is generated for every object. Checking whether the mouse is inside the square must be done using special code inserted using one of the code insertion points. When the mouse is indeed inside the square we change the color depending on which player's turn it is.

Now all that is left is defining the world, which is simply adding the nine squares to the world in a grid-like fashion. We use objects based on the object definition above. Here we overwrite part of the shape definition of the parent, by giving the location the shape should be at. The size of the shape and the methods are inherited from the object definition. This allows the squares in the world to be defined with only minimal code per square. This inheritance is a typical object-oriented feature. Object definitions can also inherit from each other, creating an arbitrary tree-structure containing all objects and object definitions.

Implementing the check for the win condition is a little involved, and not shown in the source code. You would have to create a win-event and let the squares in the world trigger these by checking whether the color change to that square makes up a row of three consecutive colors. Then some objects can be created that somehow show that the user has won. These should react to the generated event.

3.4 Game XML

Taken from [10]:

”GameXML is a collection of XML specifications which describe and script computer simulation engines. Developed by the XML Game Consortium (XGC), it is an on-going project to create a reusable, standards-based architecture that can be applied toward computer games and simulations.”

At the moment the XGC maintains two game languages. One language is aimed at simple board games (GameXML/ABG), while the other is aimed at the more elaborate domain of role-playing strategy games (GameXML/RPS). Both examples might prove useful as they contain different aspects of gameplay and game programming that might be needed for a general computer game DSL. Unfortunately, GameXML/RPS is a language still under development and not a lot is known about this project. In section 3.4.1 we take a look at the different languages that are part of GameXML and their interrelations. Section 3.4.2 gives information on the language on which the board game language is based: the Zillions Of Games format.

3.4.1 Different Languages

GameXML uses a model-driven engineering approach to the domain of 2d computer games. They start out with the MORPH meta language [51], which allows for the specification of (domain-specific) languages in XML, several languages are then defined using MORPH. A level lower in the hierarchy are the domain-specific languages that are intended to ’implement’ game domains (e.g. board games, real-time strategy). The next level contains the game system, which is a system in which the final game can be created (e.g. chess system). The last level is the level at which the actual game is defined (e.g. chess). This is depicted in figure 3.4.

GameXML gives a couple of languages that can be used as a support language for both the process and the domain languages. These are:

XRealm : a language for describing storage of digital assets

XTheme : for the definition of the digital assets used in the game.

XScene : allows for describing 2d interactive multimedia interfaces

XLobby : a language which allows for describing communication in a collaborative environment

Using the GameXML approach two languages are being created, one of these languages is aimed at describing board games, while the other aims to describe role-playing strategy games (a mix between strategy and role-playing games). The language for board games is based upon work done by the Zillions of Games application, which we describe next.

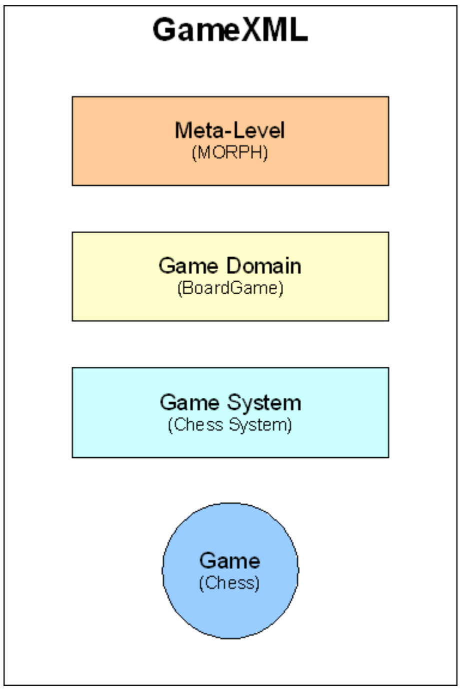


Figure 3.4: GameXML overview

3.4.2 Zillions of Games

Zillions of games is a game package with a 'universal gaming engine' for board games. This allows players to play nearly any abstract (2d) board game or puzzle. This engine takes as an input games written in a specific language for board games. Since this is a prime example of a domain-specific language I will give an overview of the most important language features.

The language

A game defined using the Zillions of Games language format (ZRF) consists of four basic parts. Each of these parts will be discussed in turn. The first part is the *game description*, which contains the title of the game, a description of the game, some history, and some strategy information. This is interesting meta-information, but can be considered as not important for this study.

The second part of the ZRF file contains a definition of the objects in the world: the *pieces* and the *players*. Multiple pieces can be defined, including their look for each player, some meta-information, and the moves allowed for that piece. The pieces must then be places on the *board*, which is the third part of the ZRF file. The board is given a shape, dimensions and a look. The initial layout is specified using the pieces defined before. The last part of the ZRF file is used to determine the *win-or-lose condition*, this is a rule defining when a certain player has won or lost the game.

Zillions of Games recognized the need for a different language to define AI modules in. Conventional programming language are better suited for this purpose than the

Zillions of Games language. Therefore a plugin framework has been created in which authors of a game can provide a special AI module to perform the AI for their specific game, while allowing the game to be specified in the Zillions of Games format.

The language also has support for macros that ease development. Shortcuts can be defined up-front to prevent repetitive typing when using the same concept, in effect you can extend the existing language with more specific purpose constructs. Rather than being imperative in nature, the language is declarative in nature, this helps using the new constructs in an effective way.

Tic Tac Toe

Applying this to the Tic-Tac-Toe example will mean that we have to define the different pieces and players, in this case two players and only one piece (behavior is the same). Next we need to define the board, which is basically a square with a 3x3 grid superimposed. The win and draw conditions are defined by specifying the board configurations which produce that situation. The keyword 'stalemated' indicates that no more legal moves can be made by either party.

Appendix C contains the complete source code for the Tic Tac Toe example. For extra help when trying to understand this code I refer to the Zillions of Games language reference [38].

3.5 File Formats

Looking at different file formats that are being used in the games industry and supported by the games industry might uncover interesting details about the current way of thinking (paradigm) of game developers. Since we want the DSL to conform to the game developers, and not the other way around, taking this into account for defining what the DSL should cover and what parts can be distinguished is very important.

Two file formats that are currently being supported by the industry are Collada [22] and X3D [8]. The first of the two is actively being developed by the Khronos Group [23] and is the primary file format for the Playstation 3 [48]. Collada is promoted as an open standard which allows for the exchange of digital content between software packages. X3D is rather similar except that it is aimed at providing interactive digital content through the web and for more 'serious' applications. In the following two sections we will describe these file formats in more detail.

3.5.1 Collada

Collada is a collaborative design activity to establish an open standard for sharing digital assets between interactive 3d applications. Collada was adopted by The Khronos Group when version 1.4.0 was released [24]. The high-profile members of the Khronos Group promote Collada as the main standard for digital assets sharing through toolchains in the industry of 3d interactive applications.

The Collada schema supports the major features that all modern 3d applications share, among which game engines for games. A Collada file is made up out of several libraries of different types of assets. These assets can then be used to build up a library of different scenes. One scene can then be selected as the active scene. Collada is also separated into three different parts: the core, a physics part and the FX part (containing special effects).

Core

The Collada core contains most standard visual assets. It has a geometry library for storing the different geometry available in most 3d applications today. It also allows for the specification of lights and cameras, which can later be used together with the geometry to create a scene. Images are also included to allow for complete scene information to be stored.

Combining geometry, lights, cameras and images allows for the building of scenes using the scenegraph principle. Parts of the scenegraph can be stored in a library and used to prevent code duplication. The scene can be build using information from the libraries or by using newly created elements. It is also possible to store animations, controllers and other data related to animating the scene or specific objects in the scene.

Physics

To provide physics support there is a second part of Collada specifically created for this purpose. This part allows support for defining rigid bodies, physics materials and special physics scenes which define the world based on physics object. These physics objects can then be used to control different objects in the visual scene constructed in the core part based on some physics simulation.

FX

The FX part of collada allows for defining complicated shader effects to be build inside collada. These effects can then be applied to objects by defining different materials using the effects. Materials can differ in the details of the parameters or the textures used to render the objects using the effect.

3.5.2 Extensible 3D (X3D)

Extensible 3D (X3D) [8] is the successor to Virtual Reality Modeling Language (VRML). Unlike VRML, X3D is based on XML [9], which makes it easy to read and understand. It basically uses the same approach as Collada: provide a common profile with the basic functionality and allow extensions. Unlike Collada, the language is not only aimed at providing a file format for storing assets, but it also allows for specifying run-time descriptions, such as interaction with the user. To allow run-time descriptions X3D has an event model and a scripting API.

User interaction is done through the use of events. Objects can have sensors attached to them that give an event when something specific happens. For example when the user 'touches' the object with a pointing device. The event then fires and some simple behavior can be created. Using scripts more elaborate behavior can be defined. The obvious advantage here is that the behavior of objects is defined in the same file. However, considering the need for a scripting field containing regular code, the file format is not particularly suited for this.

X3D is mainly aimed at providing a format for web-based content for a variety of hardware devices. It is not particularly suited for computer games, besides needing some viewer application for the game defined in the X3D format, one would also need to do a lot of extra scripting using the API to accomplish anything useful. This means most of the work still has to be done yourself, X3D does not provide an advantage for this. A scripted part is just a piece of code inserted into the XML file, requiring an external application to be started to execute the script.

Chapter 4

Games and Game Developers

This chapter presents the results of investigating games and game development in practice. It is important to know whether the current state of work process improvements and game development insights are being used, at least to some extent, in current game development. Examining current practice in game development also provides insight into the workings of game studios and allows us to look for commonalities between games and between game studios. These commonalities and work process details can then help to further improve upon the elements of the DSL that will be developed.

In section 4.1 we will start by examining games, the basic types and the commonalities between them. We verify this by looking at their source code and the way this evolves over time in section 4.2. Section 4.3 will give the results of interviews I held at a tour along game studios. A tool used by most game developers is the game design document, this document will be discussed in section 4.4.

4.1 Different Game Types

Before looking at how games are developed and what tools are used, it is important to know which types of games are available and what their characteristics are and what assets and elements go into creating a game. Looking at these different games will give insight into the scope of the domain and the different elements that need to be part of the DSL.

4.1.1 Main Game Genres

Before we start looking at the different games and their respective source code it is important to first consider the different types of games that are being produced. Game types are usually given by the game *genre*, which distinguish the type of gameplay from the other. Genres are mainly focused around the style of interaction [60] and thus provide a good basis to find out whether the interaction type influences the general structure of a game. In this section we will give an overview of the most important game genres and compare these to find common elements and differences.

Here we present a high-level overview of the different games genres [43]. The most important game genres are summarized in table 4.1. When looking at this table their is a surprisingly little number of really different game genres and most of the games encountered will fit into one of these genres (sometimes a combination).

Genre	*Perspective*	*Objectives*
Action and Adventure	Usually from the character	Overcome physical challenges, puzzles, races, and a variety of conflict challenges. First-person shooters are example of action games
Strategy	God-like, controlling	Tend to include strategic (naturally), tactical, and logistical challenges, in addition to the occasional economic ones. You usually have to mobilize resources to organize something (army, economy, etc).
Role-playing (RPG)	From the protagonists	Usually involve collecting loot and trading it in for better weapons and equipment. The main objective is to work through a story and upgrade your characters.
Real-world simulations	Usually first person, but depends on simulation	Include sports games and vehicle simulations, including military vehicles. They involve mostly physical and tactical challenges.
Construction and management games	God-like, controlling	These games are primarily focused on developing businesses and organizations. This genre is rather similar to strategy.
Puzzle games	Overview of puzzle	Tend to be variations on a theme of some kind. Sokoban is about moving blocks around in a restricted space; The challenges are almost entirely logical, although occasionally there's time pressure or an action element.

Table 4.1: The main game genres

When we look at the differences between these genres, the most obvious difference is the way the view is constructed, in some genres the perspective is first/third person, close to a player and linked to a player. These perspective also imply that the player exerts control on the world through this character. A second type of game gives a view from the top or some other overview perspective. In these type of games it is usually implied that the interaction with the world is through some god-like possibilities of control (i.e. units or buildings are magically instructed to start doing something).

Considering these differences it is important to realize that these are only differences in the view of the world and the specific way of interacting with the world. In its purest form all the game genres described above feature some kind of world, objects and characters in this world with different kind of properties and behaviors and the rules that make up the game. Since these are typically elements that occur in every game genres, we would not have to worry too much about the game genres for defining the general structure of the DSL. Considering the game genres when it comes to specific ways of defining interaction, behavior and properties of objects it might be necessary to provide some specific features to support all game genres.

4.1.2 Elements of a Game

According to an overview presentation of XNA [31], containing the results of detailed research in the game industry conducted by Microsoft, there are several important elements when it comes to game design. The team behind XNA has separated the elements into three pillars: process, game technology and design/content creation, we will focus here on the latter.

When we look at this last pillar, it is interesting to see that the gameplay itself is completely omitted. XNA focuses mainly on the different assets that can be created, without considering gameplay and game design as assets. The elements they do present are:

- Meshes and Materials
- Audio
- Cinematics
- Animation
- User Interface (not really aimed at interaction)
- Worlds / Levels

4.1.3 Game Design

Besides the 'visible' game aspects discussed above, there are also some 'hidden' elements when it comes to implementing the game using code. We will now focus further on the different elements that go into designing a game. If we take a close look at the current insights in game design [43], we can distinguish four elements that are of importance to the game design and the implementation:

The interaction with the player determining how to control the game and what feedback is provided. (This is given shape by the visual assets discussed above).

The objects that make up the game world, with specific behavior and properties.

The rules that govern the core mechanics of the game and determine what the player can and cannot do.

A storyline is not present in all games, but in some genres it takes a prominent place (action & adventure, RPG).

4.2 Source Code

Now that we have an overview of the different games around and the elements that make up those games, I will check for the presence of these elements by looking at the source code of games. Game source code could prove another valuable source of domain knowledge [46].

Section 4.2.1 discusses the Quake game series, an influential series of games which has had its source code released to the public. In section 4.2.2 I will take a look look at game source code of student games while the game is being constructed. The changes in the source code are discussed and the most important components are summarized.

4.2.1 The Quake Series

The Quake series are developed by id Software and is considered to be ground breaking by many people. id Software pioneered many of the common modern game-play elements. id Software was (one of) the first to come up with a 3D-ish game ’Doom’ [34], which changed the look & feel of games forever. Besides viewing the world in 3D it also provided a whole new way of navigating and controlling the world which remains practically the same to this day. They effectively invented 3D gaming and a whole new genre: First-Person Shooters (FPS).

Quake is a game which looks a lot like Doom in the respect that it has the same genre (FPS). Up until now the source code of the first three Quake games have been released to the general public. This allows for extracting common constructs and semantic notions from this source code to arrive at the important components of a professional game series.

The source code of Quake III Arena can be broken down into three distinct parts:

- The *game*, which really governs the game (on the server),
- A *client* portion of the game which handles part of the interaction with the user,
- A *user interface* part which governs the remaining interaction.

These parts correspond quite well to the elements of a game recognized in section 4.1.2. However, the rules and object code managed at the server still contains a lot of technical details which makes it harder to understand and maintain the code and to predict what the potential impact of a change might be.

The code is also rather unstructured in this sense, making it hard in general to change specific elements in the game. Since the game design, rules and interaction are so intertwined in the game it is hard to concentrate on one of these aspects when making a change. It almost always will affect one of the other elements as well, making it harder to maintain the code base.

4.2.2 TU Delft - Games project

The bachelor curriculum of 'Media- & Kennistechnologie' (Media and Knowledge Engineering) at the TU Delft contains a project which is aimed at developing a computer game in a very short amount of time (7 weeks, 8 hours a week). The Cannibal Engine has already been used twice as the supporting middleware for this project and will be used again in the 2006/2007 version of the project.

The games created during this project are, of course, very basic and only contain the bare minimum elements of game play and graphics. By investigating the source code of these games we will be able to concentrate further on the different base components of a game. Since the games industry is quite secretive, these games are also the only opportunity for me to get hold of source control repositories to analyze changes over time. Since these are small games, the use of such an analysis is quite limited, but some information might be gained as to what parts are most difficult to implement.

Looking at changes over time for these projects provides insight into the different elements that get changes throughout the projects and the different elements that require a lot of programming work without directly being useful. By looking at the different changes we can therefore draw conclusions as to what elements of the DSL should be implemented very explicit and flexible and what elements the DSL can abstract from.

Red Ribbon Rabbit

Red Ribbon Rabbit is a game that would fit into the adventure genre. The player is represented by a rabbit. The object of the game is to gather carrots, evade the wolf and solve some simple puzzles. One of the major accomplishments of the group developing Red Ribbon Rabbit was that they had a very basic working prototype pretty early in the development process, which allowed them to focus more attention to designing new game design elements which improved the game and made it more fun to play. Due to this accomplishment, the version control repository should reflect this focus by giving an overview of changing game design elements through time.

When looking at the changes over time for the source of Red Ribbon Rabbit it becomes quite clear that a solid design led to a very early first playable, which allowed them to start working more on changing elements of the game design. Unfortunately, changes to the game design were not that easy to implement. Having a solid design helped, but still code had to be changed in several places and most of the code dealt with technical issues. Trying to avoid these technical issues would not completely be possible, since they are usually quite specific, however gameplay code is intermixed with technical issues and mathematical formulas, making the code harder to read and understand.

Even with a solid design is was necessary to adapt the code many times to fix errors and to implement the changes in the game design. Doing this, the quality of the code rapidly became worse, cleanliness of design and clearness of code was often traded against speed and features of the game. The more features and decoration was added to the game, the worse the architecture became.

A language which would give more structure to the game design aspects and a solid way of connecting this to the implementation of the representation and interaction of the user would have kept the code more clear. This is an area where a DSL can most certainly help.

Lightmare

Lightmare is a game that would also fit quite well into the adventure genre. The player is represented by a walking lightbulb with two batteries attached. The object of the game is to find the correct power socket, which provides power to the light. This has to be done while evading malicious hammers and various electrical devices stealing power. Unlink Red Ribbon Rabbit, Lightmare was defined quite well to start with and all features were implemented all at once leading to the fact that the game was only starting to work in the last week. This is a different way of development and it might be interesting to see how this manifests itself in the source code changes over time.

When looking at the source code of Lightmare we can see that it has a design which reflects the game design aspects quite well. However, the team behind Lightmare did not manage very well to tie this to a working implementation. They got caught up in (technical) details which led to a first playable very late in the development process. Game design aspects could hardly be paid attention to, while these are usually considered the most important aspects.

A special language which would have provided support to take the step from defining the game objects to defining the interaction with the user, could have helped this team to resolve these issues. The rules of the game are also scattered throughout the code. Capturing these together and having them clearly defined, without technical details, would help to keep more structured code.

4.3 Interviews

In this section the results of the interview tour are presented. In the following sections the results for the most important studio types is discussed. Before the results are discussed, it is important to explain the reasons for the interview and the way the questions were devised.

For the purpose of designing a DSL that closely resembles the way of thinking employed by game developers, it is important to get to know the mindset of game developers and the way in which they work together at game studios. This 'tacit' knowledge [19] is hard to find by looking at the results that game developers produce: assets, source code and the final game.

Extracting the way of thinking from these results is difficult, there is not necessarily a direct link between the results and the way of thinking. The way of working is even more difficult to extract from the results, even if the way of thinking can be found, the way of working cannot necessarily be extracted from that. To do get this information it might prove useful to look at game source code while it evolves: version control repositories that show changes over time allow for more insight. But, since game studios are quite secretive about what they are up to, they are very reluctant to releasing this kind of information.

Besides looking at source code, which on its own may lead to false conclusion or a lack of conclusions, it is a lot more direct to interview game developers and to figure out their way of thinking and working by listening to there reactions. The interview is constructed using techniques from qualitative research methods [44]. The interview is focused mainly on the different persons in a game studios, what their job is and how they cooperate. Another important element of the interview is projective techniques, where the respondent is challenged to reveal underlying motivations, beliefs, attitudes, or feelings towards the interview subjects. Because of secrecy reasons it is not possible to give the answers to the questions directly, and I cannot disclose the interview partners. The results are therefore anonymous and aggregated.

4.3.1 Setting up the Interview

The interview is set up to be part of a bigger interview in which game developers are interviewed on a variety of subjects, ranging from composition and size of the game studio to the work process used and the individual tasks and way of thinking of employees. This larger interview is used to gain a more complete picture and to support multiple research projects currently active at Cannibal Game Studios.

The results of these interview sessions may not be generalized to all game developers and game studios, so it would be useful for future work to do a more elaborate interview tour, including game studios from the USA, UK and Japan. However, since the results of the interview are quite in line with other movements in the game industry: middleware discussions, file formats and the related work presented in chapter 3, it seems quite reasonable to accept the results as general conclusions.

The interview contains some general questions aimed at retrieving information about the company (its size and composition), the way in which collaboration is handled with the company (e.g. version control and project management) and the general work process within the company (e.g. content pipeline and bottlenecks). Besides these more general questions there are some in depth questions concerning current middleware and engines used, the advantages and disadvantages of these tools and what game studios expect from middleware.

The part of the interview that is most interesting to consider in light of the DSL to be developed are the questions about game design. The questions start out by asking the game designer to specify a game. This not only provides insight in how the game designer thinks, but also to what he considers to be part of a game design. Then some questions are asked to compare the current methods used by game designers with the way he thinks and reasons about games. A third set of questions is aimed at the workflow, communication and collaboration within the domain of game design. This allows for a more detailed overview of these types of considerations for the design of the DSL.

4.3.2 All-round Game Studios

All-round game studios are game studios that have the entire game production in-house, not only the game design and the content for the game is created, but the code is also written by developers at the studio. This not only enables better control, but it also means that the team members can work more closely together, which introduces special demands on the middleware used.

Typical game teams in this category range from 10 - 150 members, depending on the quality and size of the title produced. Depending on the type of project and the available middleware the ratio of artists to programmers range from 70/30 to 50/50. Communication is usually done through experienced leads that divide the work and create a project planning.

Collaboration between programmers, artists and game designs is roughly as follows: first the game design is created and captured in design documents. These design documents reflect how the game should look and work. The programmers and artists then start creating assets and source code, these should be combined into the final product by using middleware products. In most middleware products this is done by creating models and animations in some third party modeling package, importing these in some level editor and then packaging this into the final game. Using the level editor and importing objects (both the graphics and behavior) is usually done by level designers, who design the game on a lower-level than the lead game designer.

4.3.3 All-artist Studios

All-artist studios are game studios led by visual and gameplay specialists without any technical knowledge. These studios sometimes makes extensive use of freelancers which do small parts of the content creation or design work. Technical implementation of the game is usually handled by separate technical companies. At any one time there are a lot of different people doing active content creation or design work. In a sense these studios are unexperienced when it comes to technology, which provides them with a unique unbiased perspective.

Collaboration with the different freelancers is usually done by the use of off-the-shelf forums and wiki. These slowly get adapted until the tools have become an intranet collaboration tool in which users can login and manage their own tasks and contributions to the game. It is most important that everybody knows what they are working on and what others are working on, especially when teaming up with technical companies that are supposed to implement the game.

Most of the all-artist studios feel that the programmer is the one that finally brings together the game design with the content created. It is the programmer who integrates the two and who might make it possible to add content to the game without programmer intervention by creating restricted frameworks and tools. Game design elements are still to be implemented by a skilled programmer who has read and interpreted the design documents.

4.3.4 Content Creation Studios

Content Creation studios are a lot like the all-artists studios described above, with the difference that they also have, at least some, technical knowledge. These studios create content for other games, by applying their artist expertise with their technical knowledge. They normally do not create games from beginning to end, but small parts of it: levels, type of asset (trees or buildings), etc. Some all-artists studios use content creation studios to create some assets for the game, and the other way around. Because more knowledge is bundled in these studios, they are usually a bit larger than all-artists studios.

Since content creation studios usually work closely together with multiple other companies it is important that they can collaborate well. Therefore these type of studios usually have a lot of knowledge when it comes to collaboration. However, while some work together in novel ways, the technology they use does not reflect this. Tools are not integrated well and many different systems are used to manage projects. Integration of different content pipelines is even a bigger issue. The collaboration, from a technology viewpoint, is not very much different than the studios discussed above. These studios usually use the same methods when it comes to specifying the game design, either no real method or just using a design document.

4.3.5 Academia

The Netherlands has some research and educational activities when it comes to computer games. The most noteable are perhaps UPGEAR [3], a collaboration between Utrecht University (UU), Utrecht School of the Arts (HKU) and Hogeschool Utrecht, and the Center for Advanced Gaming and Simulation, a collaboration between UU, HKU and TNO. These collaborations are to give a boost to education and research in the area of games and simulation. An example of a focus point for these groups would be combining motion planning with procedural animation of natural moving characters.

It seems most academia have a rather classical view of the game industry, which lacks a bit behind the actual developments. An interesting thing to consider is the use of the game design document: academia see this as the bible of a game. First the game design document should be setup, using prototyping where it is possible to test concepts and balance the game. If changes need to be made they will first have to be considered against the game design and then implementation should be judged.

An interesting thing that academia seems to focus more on than the studios itself is the change of emphasis for games. The emphasis has been on the looks of the game; the graphics should be cutting-edge. However, as the graphics become more and more realistic, other aspects of the game are becoming more important. These are especially the game design aspects (e.g. gameplay, AI). Problems are anticipated though, since there are no real standards (yet) for specifying and implementing these aspects. Creating this DSL might help in this aspect.

4.4 Design Document

One tool that all game developers use when communicating information about the game to all different disciplines involved is the design document, or a set of design documents. I will discuss the design document here along with the way it is used across different studios. Most types of design documents contain a couple of elements that are always present. These elements are interesting to discuss, as they are the core elements of what a game represents for game studios.

An important part of any design document is a *one-pager*, describing the game in one page. This page should include all the elements of the game in a quick overview style. It should contain the setting and basic story of the game, the important game mechanics and some basic objects (e.g. weapons) and characters. Mostly this one-pager is used to get outsiders up to speed on the project and to try and get them enthusiastic. The one-pager is therefore also considered as a separate document.

Traditionally the game design document is a long document containing full details of all aspects of the game and designers had a big task keeping all details corresponding to the actual implementation of the game. The game design documents are now becoming more like Wiki [35] pages, a set of web documents which anyone can easily edit and change.

Not all game studios use the design document in the same way. Some studios have one or two game designers who are responsible for sculpting the entire game design. This requires them to create a highly detailed description of every part of the game to have everybody share the same ideas. Some studios choose to only sketch out the main features and the setting of the game, allowing everybody to add to the game by coming up with new ideas and filling in the blanks with their own creativity. Usually this collaborative process is guided by an experienced game designer who holds meetings often to synchronize ideas and make decisions.

In either way mistakes and omissions will happen when creating the game design document. This means that the design document is constantly changing, explaining the use of and need for dynamic systems like Wiki. It is important to know which aspects there are to a game design document to determine the elements that can get changed throughout the game development process.

One of the most important parts of any design document is the part about the game mechanics. This part explains the most important game play elements of the game. It describes the features of the game play (e.g. jumping from platform to platform, shooting enemies), the flow of the game (i.e. similar to use cases), some information on characters (e.g. special abilities), physics and ai, statistics (e.g. score) and other aspects of the gameplay that are important to the users experience.

A second aspect of gameplay that is included is detailed information on every character, enemy, weapon and other object in the game. All of these require descriptions that allow other members of the team to work on them while all know what its purpose is within the game.

Besides the gameplay elements, most games have a storyline or different levels throughout the game. A second section of the game design document gives details on the story of the game, the levels the user has to go through and sometimes explanations for the gameplay elements. For example, if characters in the game can jump incredibly high this could be explained through some story in which the gravity of earth is distorted by some event.

The story not only guides the player and motivates him or her, but it also provides a setting for the game. Using the same gameplay mechanics one can create a game in which the player shoots at invading aliens with a lightning gun [17], or a comical game in which you have to throw snowballs at mutated turkeys [28]. Both can contain exactly the same gameplay.

The setting of the game is also supported by the art work, the video and the sound effects and music used throughout the game. All game design documents contain some details on how the game should feel, what the overall mood is and at least some impression sketches or concepts art of the game. Using these, guidelines are created to which the artwork and sound should adhere.

Besides defining the game mechanics, the story and setting, it is also important to think about the interaction with the user, most notably controlling the game. For some games this might be by controlling one character directly, but others have a more god-like control in which the players control an entire organization or country. How the user controls the game world is an important aspects of interaction.

Another important aspect of the interactions are the heads-up-displays or other GUI elements within the game, like menus and help screens. These mostly define how the user controls the system surrounding the game: saving, loading, (re)starting the game, changing some settings, etc. These are usually described using classical storyboard techniques: creating flowcharts, creating screen mockups and defining functional requirements.

Some game design documents also contain details about the technical details: the engine used, what AI system, what lighting models, etc. However these are again implementation details that the game designer should not have to think too much about. Of course they are important to keep in mind, however a lot of time is lost thinking about these details.

Chapter 5

Cannibal Game Development Platform

As mentioned in section 2.3, an important step in developing a Domain Specific Language is constructing a support library for the different operations and semantic notions of the DSL. The DSL to be developed during this master thesis was meant as an extension or add-on for the Cannibal Engine. However, during this research some interesting facts and ideas came about that dramatically altered the Cannibal Engine. These changes led to redefining the Cannibal Engine as the Cannibal Game Development Platform and making some changes and additions to the existing code.

In section 5.1 an overview will be given of the findings that led to the changes to CGDP, the changes itself will be presented in section 5.2. The findings can be considered as a short summary of the important points from the previous chapters relating to the platform, while the changes can be considered as an overview of the features implemented early-on to prepare the platform for usage with the DSL and to supports its main paradigms. It is important to note that the changes described here are only the changes that are made to the core engine framework, not additions to the platform in the form of a run-time system for the DSL.

5.1 Findings

This section gives an overview of some general findings that were encountered during the research into a DSL for computer games. These findings have led to some (radical) changes to the Cannibal Engine, which has since been called the Cannibal Game Development Platform.

In section 5.1.1 I discuss the implications of applying the model driven view introduced in section 2.1 to computer games. A more user-centered approach, highly recommended and hinted upon by the game studios in section 4.3, is discussed in section 5.1.2. When looking at related work and existing tools discussed in chapter 3 it becomes clear that some elements are important candidates for implementing in a DSL. The elements are discussed in section 5.1.3.

5.1.1 Model Driven View

If we look at a computer game as a model [32], we can distinguish a very diverse number of components that need specifications. The rules of the game, special graphical effects, the look of the game, the interaction and other details need to be specified. If we take a broader look at these different components of a game we need to realize that not all of these components can be specified in one language. Even if we could do this, it would not be the desired approach as will be explained further in section 5.1.2.

If we look at sections 4.1.2 and 4.1.3 we can distinguish very different elements of a game that determine the look and feel of the game. These different elements all have an important role in the overall design of the game, but they are usually constructed and created by different people and in a different way.

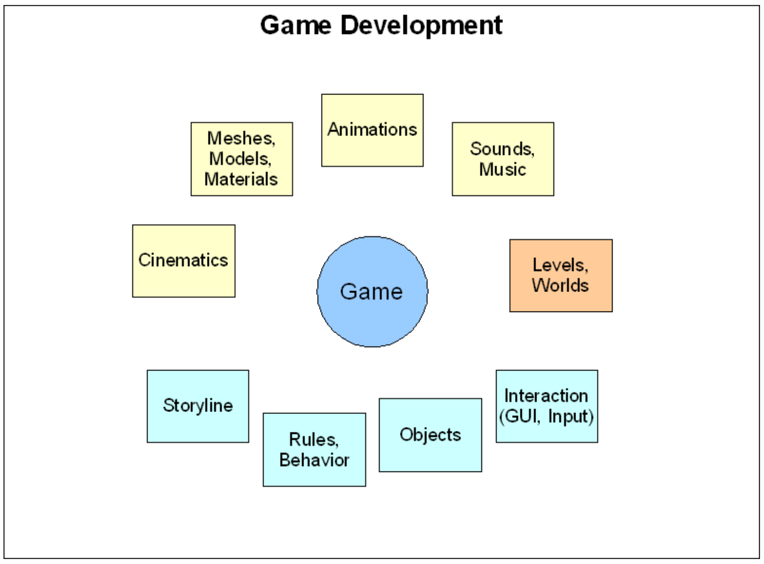


Figure 5.1: Overview of game development

Taking into account the diverse nature of these components and the solutions that already exist for some of these components, the remaining part of this master thesis will mainly be focused on the objects in the world, their properties and behavior, the rules of the game in a DSL and the interaction with the user. These are elements that are the primary concern of the game developers along with game-play designers and programmers. There are already existing tools for specifying the composition of the world (locations of objects) and for the look of objects (3d models, textures and shader effects).

5.1.2 User Centered

During my research into the specific needs and way of working of game studios it became apparent that the available middleware that is being used in the gaming industry is not always geared to the people working with it, or in other words: not *user-centered* [37]. A lot of game studio team members are still being confronted with details they should not be confronted with: a game designer will only want to be working on and thinking about the rules of the game and wants to express these ideas directly into the game.

Right now the game designer will have to express its ideas in some format that programmers hope to understand (the game design document). These programmers will than have to interpret those ideas and try to express them into some programmer environment, usually not particularly suited to expressing these ideas, being bothered will all kinds of implementation details at the same time. Given the number of steps involved a lot of errors can be introduced, and this is not a desired situation. In the ideal situation a game designer can express his ideas directly to the game development environment in a language close to the designers mental language.

The same logic applies to other responsibilities and team members. For example in some studios it is still the case that the artist must work in close collaboration with a programmer to create shaders to attain certain graphical effects and looks. This obviously leads to another set of steps of interpreting and re-expressing the original idea in other languages, while having to account for details that should not matter to the artist.

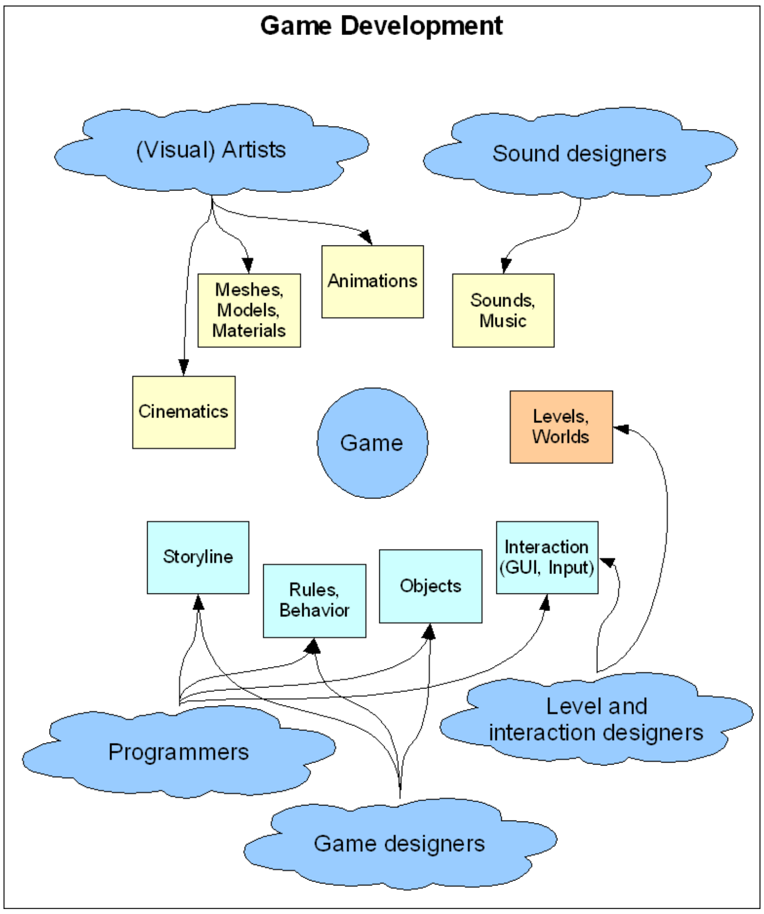


Figure 5.2: Overview of game development - continued

In figure 5.2 we can see that the different usergroups have been added that are responsible for creating the specific assets (see section 4.3). The usergroups are very aggregated and normally they consist of a team of people in which different responsibilities are divided and some hierarchy is present.

The tools available for the four content items at the top-left are already well established and used throughout the industry. These tools are really specific to their purpose and work well for the person that actually uses them. Levels and worlds usually have an editor which is given by the developers of the engine (e.g. UnrealEd), these editors somehow allow for importing the other assets, objects and behavior to assemble the levels. For the four bottom assets there are really no specific tools available for creating them. These four assets are currently implemented using some game-specific framework based on general-purpose languages (GPLs) or by using GPLs directly.

Besides being created in tools that are not particularly suited for that purpose, the four bottom assets are also created and heavily influenced by two completely different user-groups. Of course, game designers have to design the game play and core mechanics, but the programmers, will have to implement their ideas and make it reality. This leads to a rather entangled situation where the different assets are controlled by two parties at the same time. It would be highly desirable to better separate the aspects of programming and game design to provide both user-groups with more freedom and flexibility in their work.

Of course it is important to realize that all the assets depicted are interrelated in some way. The objects defined by the game designer somehow have a representation with graphics and sound defined by the artists. The storyline has a strong tie with the cinematics and animations and the rules partly determine the type of interaction. Levels bring all of these elements together by modeling the actual world that the game will take place in.

5.1.3 Elements of a Game Design

When looking at the game discussed and the work that has already been done in the area of a computer game DSL, we can conclude that there are some rudimentary elements that are always present in any game design in general. These are also the elements that are supported most poorly by available tools, if there are even any.

The elements that will be the focus point of the game DSL are highlighted below for clarity:

Objects need to be defined in some way: their properties and behavior should be specified. An object-oriented style for doing this is most likely the best solution.

Rules need to be defined to determine the boundaries for the game and the actions that specific actors in the game are allowed to undertake.

Interaction with the user should be specified in some way, this can likely be done most naturally with event-driven techniques.

The Storyline should be defined for games that have a storyline, a linear storyline or tree-like structure can be used best.

Of course it is important that it is clear how these elements are interconnected and attention must be paid to make sure they can be connected in an intuitive way. As mentioned before, interconnection with other assets is also important to assure that everybody can work together effectively while focusing on their own expertise.

5.2 Changes

The findings above led to a couple of dramatic changes to the Cannibal Engine. Cannibal Game Studios decided, based on the findings, to redefine the Cannibal Engine as the Cannibal Game Development Platform. This section will explain what the difference between these two names means in practice and what changes the DSL has instilled upon the core engine framework. Some features will be left as part of the language run-time, while other are changed natively in the engine.

Until recently, Cannibal Game Studios was building an engine with some surrounding tools to help developers write their games and help them by providing solutions to common problems and by abstracting away from low-level system calls and from hardware details. The different findings led to a shift in focus for the Cannibal team: the focus is now more on eliminating tasks the user (the developer) does not want to be confronted with and enabling them to do the work they want to do. For example: an artists will only want to be working on the look of a certain 3d model and it's material. He does not want to deal with all types of different hardware details and limitations of shaders. A game designer on the other hand will only want to think about the structure of the game and how the different elements together form the game-play, not about specific implementation or on how to convey his ideas to other team members.

5.2.1 Overview

The shift in focus has led to a number of radical changes to the development of the engine. First the *paradigm* (or way of thinking) has been developed in collaboration with different game studios in the Benelux region. This means that the responsibilities and mindset of every function within game studios have been mapped. These responsibilities and mindset are then used to determine the general paradigm for developing games. This paradigm includes the personal way of thinking of team members and a total process that can be used to develop games with the Cannibal Game Development Platform.

This paradigm is used to derive the tools and languages (DSL) that are needed to enable developers to work in an efficient manner and to not be bothered by details that do not or *should* not concern them. The DSL will be used along with the tools to develop games; both are of equal importance. This implies that it is not needed to have the computer game DSL support all aspects of game creation. Some aspects can be represented better by different tools or other means of development. For instance the writing of a plugin for a certain graphical effect can best be done using conventional methods (shaders), which are specifically developed for this purpose and used throughout the industry. Specifying the look of models can best be done using conventional DCC tools artists are comfortable with and maybe some support tools to link the graphical effects to the 3d models.

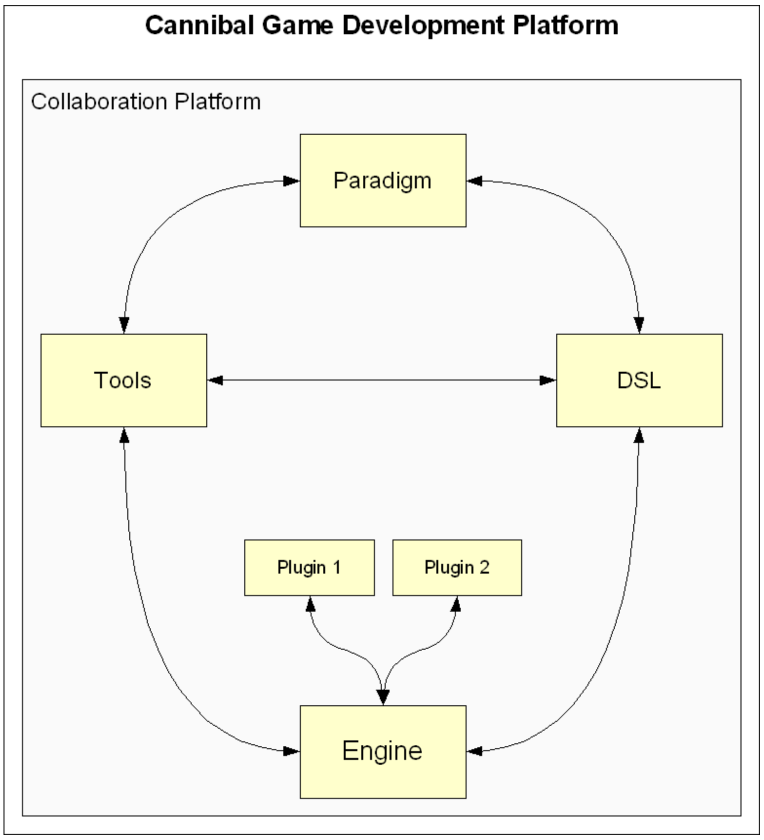


Figure 5.3: Cannibal Game Development Platform Overview

As already anticipated in section 2.1, a DSL for games should be highly extendible; this also goes for the entire platform. Many game developers have different needs and requirements, it would not be feasible trying to provide every single piece of functionality to clients up-front. Rather it should be easy to extend the platform where necessary by providing *plugins*. This is depicted in figure 5.3. To serve this purpose, a special support framework has already been implemented through which any plugin can provide the platform with necessary extensions.

5.2.2 Events & Actions

Another important realization is the fact that game development tools are moving more towards an event-driven, action-based programming model (see sections 3.1.3 and 3.2). In other interactive application areas, like simple graphical user interface systems, this paradigm has been used much more explicitly for quite some time [42]. Since Cannibal Game Studios also believes that this paradigm is more suited for real-time interactive applications, like computer games, the necessary support for working with events has been implemented even more clearly in CGDP.

A framework for working with events and actions has been implemented directly in the engine early-on to see how developers use this concept and to allow them to work using an event-driven programming model. The framework allows for events and (re)actions to be connected in an easy to use fashion. Further composition of events and actions using operators and other trickery is also supported natively.

Many of the key functionalities of the engine have already been exposed to this event-framework. Core engine events are exposed through this interface, but also the user input has been exposed. Any input device can expose any number of events it wants based on the input generated by the user. All components and plugins of the engine are supposed to expose events that may be useful to the developer.

5.2.3 Future changes

In the future some others changes might be made to CGDP which will facilitate other elements of the DSL natively. Elements that are already recognized as important parts of a game design, and therefore important parts of the DSL, which are not facilitated, are the rules and the storyline. The object definitions are supported by the object-oriented language Cannibal is written in and the interaction is supported by the events and actions framework. Some parts of the rules are also supported by this framework, some rules might need special facilities though.

Creating a story is a task which involves setting up a storyline with different important events that tell the story. This storyline is sometimes also depicted with a tree, or even a graph, allowing for multiple versions and even non-linearity. A storyline framework might be created to facilitate easier implementation of DSL specific constructs. However, this may also be left as part of the DSL run-time.

Some tools will also need to be developed to assist the game developer in defining these game elements. These tools present just another interface to the DSL than using the model framework directly (see section 2.5.3). However, since we want to involve the game designers along with the programmers, they might pose an easier interface to programming with the DSL to the game designers. For example, this allows for viewing the storyline not only as code (or plain text), but also as a timeline, tree or graph.

Chapter 6

Languages and theory

Before starting with the design, specification and implementation of the DSL for computer games, it is wise to first consider the work that has already been done regarding languages related to the DSL for computer games. In this chapter some similar or related languages will be discussed along with their strong and weak points, this can then be used when designing and specifying the language to improve upon the computer games DSL.

Since most of the games involve interaction and simulation, and software that relies on interaction and simulation usually work with events, we will take a look at reactive or event-driven programming in section 6.2. In section 6.3 we will take a look at existing languages aimed at providing support for multi-threading and look into concurrency theory. We conclude in section 6.4 with a discussion of multi-paradigm programming languages. But first, since the way of thinking is important for the DSL, we should answer the question of whether to use a declarative or imperative style of programming the game. This will be looked at in section 6.1.

6.1 Declarative versus Imperative programming

In programming languages paradigms there are roughly two types of languages:

- Imperative
- Declarative

Imperative languages are the more classical programming languages in which the programmer specifies which steps to execute and in what order. Using control constructs and simple statements the programmer specifies the exact steps the computer must take. On the other hand, declarative languages focus more on *what* must happen, rather than *how*. Imperative languages require the developer to explicitly specify an algorithm to achieve a goal, while declarative programs require the developer to explicitly specify the goal and leave the choice and implementation of algorithms to the compiler or interpreter.

When applying this to the DSL, declarative languages seem to fit better with the goal of abstracting further away from implementation deals. Focusing on the what is at a much higher abstraction level than specifying how (which is usually the case right now). However, it is also important to consider whether this corresponds with the thought patterns of game designers and developers. During the interviews I posed several questions in which the game designers had to explain games and game concepts, as well as answer how rules and gameplay concepts are currently expressed. It appears most of the game developers think and specify both in a declarative style. When asked to specify the game Pacman, most responded with some of the following statements:

- "The player is a yellow circular disk with a pizza slice missing"
- "The goal is to eat all the dots"
- "When the ghosts eat you, you die"
- "When you eat all the dots, you advance to the next level"

In that respect, declarative languages can be considered to be more reflective of the thinking patterns of game designers and developers and might therefore be more natural to them than imperative languages. Besides being declarative in nature, the statements above are also event-based: "when this or that happens, do this or that".

6.2 Reactivity and events

Most, if not all, computer games involve interaction and simulation. Software that relies on interaction and simulation usually work with some way of event handling. For example, in Windows actions from the user (e.g. mouse clicks and key presses) are passed to applications as messages. In traditional (imperative) handling of these actions the developer creates a loop in which the next message from the queue is retrieved and appropriate action is taken, depending on the type and the parameters of the message [42].

In the following two sections we discuss different approaches to programming based on the reactive or event-driven paradigm. In section 6.2.1 we will take a look at how modern object-oriented programming languages deal with events and event handling. Section 6.2.2 gives an overview of using events in functional (declarative) programming languages.

6.2.1 Event handlers and Listeners

An object-oriented model of event-driven programming is implemented in the various classes of the Java 1.1 AWT package [27]. In this event model, events are generated by event *sources*. These sources indicate when the event should be generated and, if applicable, with what arguments/parameters. *Listeners* can then register to these events to be notified when a source generates one of these events. This model is sometimes also called *delegation*. The classes of the AWT package let you both generate and handle AWT events.

The C# language has some similar constructs, but it allows for explicit definition of delegates (method signatures) and events. Events are defined to be of a certain delegate type, that is: they require such a delegate method to handle them. Listeners can then quite easily register to an event by appointing one of its methods as being the required delegate.

For most simple interactive applications this approach works fine. However, some applications, mostly games, require more complex interactivity and require new events to consists of multiple subevents or parts of other events. Trying to create this using the delegation model will lead to a mixture of event definitions and regular code. This makes the final code more complicated to understand and maintain, since the operations related to events are 'hidden'.

6.2.2 Functional Reactive Programming

Elliott proposes a specific solution to declarative event-oriented programming [16]. This approach uses a functional language (Haskell) to support the implementation of operators for events. This allows for simple events to be build up into more complex events, which leads to better modularity and reuse. As an example of the possibility of implementing games using a functional or declarative approach you can take a look at Frag [2, 7].

In Elliott's approach every event has a certain value type associated with it. So each event can carry some specific information with it. We will refer to this information as the parameters of the event. Considering this you can view each event as a continuous stream of value and time pairs. For instance user input could be modelled using an event which provides a value containing all potential user input. This event could then produce a value each time the user provides some input to the system.

The operators Elliott discussed include transforming an event's parameters, taking the union of two events and filtering events for some special occurrences. Other operators can also be used to create sequential chains of events or do some other transformations on events. When we look back at UnrealKismet (section 3.1.3), this is a surprisingly similar concept. UnrealKismet allows for the use of gates, conditionals, delays and sequences to do such transformations on events. Elliott adds to this by giving events a certain type of value, instead of just 'impulse'.

6.3 Reactivity and concurrency theory

Currently computer hardware is moving more towards using multiple processors to accomplish different tasks in parallel. Games are using this specific concurrency approach to do more work in the same short amount of time. Next to the central processing unit (CPU) of a computer, for years most people have used graphics extension cards to improve performance. These graphics extension cards contain a graphics processing unit (GPU), which is geared towards graphics calculations (e.g. vector and matrix multiplications). Many of these GPUs now can run the same calculations in parallel on different sets of data (vector processing). Next to the CPU and GPU a physics extension card is now available with a physics processing unit (PPU), which is geared at calculating physics simulations.

Using all of these processors in parallel might speed up execution time of games considerably. Rendering and physics simulation are both time-consuming processes, executing them in parallel will have a significant effect on the performance of games. However executing them in parallel will also have a negative effect on the performance of games. When executing several steps in parallel, in a cascading setup (see figure 6.1) the latency of the game's response to user input would increase. It would take more time for the processing of the input to reach the final output than before.

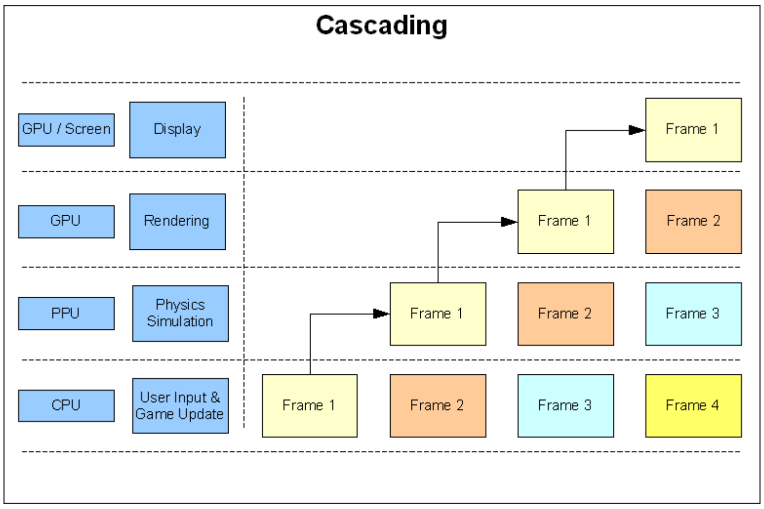


Figure 6.1: Cascading / Pipelining

The technique of cascading is a form of pipelining, where each task that is executed in order on data is still done sequentially, but each task works on a different data part. Another form of concurrency is data-parallelism, where the same task is applied to a large number of data items at once. This is what is being used by the modern GPUs. This is a parallelism that does not affect the latency, but might still speed up calculations. This might for example be applied to collision detection or updating unrelated objects.

6.3.1 Using Concurrency in the DSL

Since concurrency theory and multi-threading is hard to program and considering the fact that we want to make life easier for the game developers, it is an important consideration to leave out concurrency support in the DSL. This will make programs harder to debug and eliminate some of the advantages of using a DSL.

A manual approach to doing concurrency might be supported by a language which provides adequate support for concurrency at the language level by allowing the developer to specify processes and operations on inputs and outputs of these processes. This approach is taken by the 'Communicating Sequential Processes' language [26] and several others [4]. These languages also provide support for communication, which might also be useful for programming reactive systems; events can be considered as messages.

According to Tim Sweeney [53] only 10% of CPU utilization is spent on game simulation or game code, 90% of the utilization is spend on numerical computation.

Usually these computations are induced by game simulation code, which makes it hard to estimate the real amount of code that is executed because of game design issues.

Since the DSL for games will be developed on top of the engine, and all numerical computation is already performed by the engine at engine level, the code that is executed written in the DSL will only amount to about 10%. Therefore it is important to tackle concurrency at least at the engine level. However, it might also be interesting to see what techniques are available, as we may want to provide something in the DSL compiler to allow concurrency to be handled automatically for the game logic.

6.3.2 Software Transactional Memory

Gameplay simulation typically involves many different objects that want to modify a shared state. Each object must update 30-60 times per second to gain satisfactory frame rates. On average each update touches about 5-10 other objects [53]. It is hard to try and avoid collisions and incorrect behavior by using manual concurrency management techniques. Therefore the approach of composable memory transactions [25] can be used, this works on the following basic principles.

First we execute all updates concurrently, all updates are represented by a transaction: a sequence of statements that will be executed as one (i.e. no other statements can come in between). With a lot of updates and a small amount of objects touched per update, collisions - transactions trying to use the same data simultaneously - between transactions are likely to be low. When a collision occurs some of the transactions will need to be re-executed. Of course this method introduces some computational overhead, but if this method allows many threads to run in parallel, this is acceptable.

6.3.3 Synchronous Languages

Depending on the context in which concurrency is used, another problem that pops up when speaking about concurrency is non-determinism: the output is not uniquely determined by the inputs. When programming processes that will run in parallel, it is not always clear which process executes statements first, and this might influence the output of the code. One thing we would like to have is code to be deterministic, this makes it a lot easier to write the code and to debug it in case of problems.

Synchronous languages provide support for this by using an approach which focuses on concurrent *deterministic* subsystems that cooperate in a *deterministic* way [6]. Conventional methods for concurrency programming base their implementation on an asynchronous model of concurrency. In this model processes still compete for resources in a non-deterministic way. This effectively means that you have *deterministic* subsystems, which cooperate in a *non-deterministic* way. Non-determinism in cooperation can lead to many problems among which state problems when it comes to processes interrupting each other, views of the state for each process are not necessarily the same, etc.

The determinism in cooperations comes from adopting the rule that all reactions (sequence of statements that react to an event) are executed instantaneous: immediately (taking no time) and atomic (as one, as a transaction). Several languages adopt this approach, among which the imperative Esterel [5] and the data-flow based LUSTRE

[25]. Esterel allows for compiling the concurrent source to one fully sequential process, performing scheduling at compile-time. It can also do this for part of the source code, or for no source code at all, which means it can be very flexible as to the target platform.

6.4 Multi-paradigm

Multi-paradigm programming languages are languages that unify the concepts of different programming paradigms and create one complete language containing all these concepts. Oz [47] is an example of such a programming language. Multi-paradigm languages have the advantages of allowing developers to specify certain aspects of their program in programming paradigms more suited to this purpose than others. Care must be taken however that the different paradigms are complementary to eachother in a sensible way, and not provide more restrictions and problems. These paradigms should fit together in a way natural to the user, to prevent confusion.

Since we already recongized the need for several programming styles when it comes to defining game design aspects, it might be wise to considered a mix of paradigms to facilitate each part aspects in the best possible way. Somehow the aspects should be integrated, whether this is accomplished using multi-paradigm techniques or some special-purpose solutions with near separate languages should be decided upon when defining the language.

Chapter 7

Requirements and Considerations

Before we set out to design and implement the language and associated IDE, it is useful to summarize the requirements and give the considerations that go into the design of the DSL. The impact these have on the work that has to be done should also be considered, as this will provide us with a direction and a scope for the tasks at hand.

7.1 Language Elements

In the research part of this thesis I found that there are several different elements to game development. Particularly those related to game design are poorly supported by existing tools. Providing support for these aspects can be considered as the main functional requirements for the DSL and the definition of the domain for the DSL. These aspects are explained in more detail (by giving a more concise definition) in the corresponding chapters. The following aspects should be part of the DSL:

- *The objects* that make up the game world, with specific behavior and properties);
- *The interaction* with the player, determining how he controls the game and what is given back to the user as feedback;
- *The rules* that govern the core mechanics of the game and determine what the player can and cannot do;
- *A storyline* is not present in all games, but in some genres and games it takes a prominent place.

All of these aspects can be specified most effectively by some specific paradigm. This paradigm should be as close to the way of thinking of game designers as possible, to make specifying the game as natural as possible. This user-centered approach is discussed in section 5.1.2.

Of course the specification should be done at a high level of abstraction, getting rid of all unnecessary details. If the level of abstraction is too low we will pollute the model with complications and make it harder to bring structure to the game design.

The requirements given above indicate that the DSL to be developed consists of several parts, one for each functional requirement specified above. However, these different parts will have to interact in some way, which may lead to one multi-paradigm language or several interacting DSLs. From now on I will refer to this set of parts simply as 'the DSL'.

7.2 Focusing on the User

One thing to keep in mind during this project is that we want to make life easier for game developers and designers. Therefore the DSL should be a language that is close to their conceptual world and allows them to express their ideas without the need to translate these ideas into a general programming language.

Thus the design of the DSL should be 'user-centered' [37] (focused on the needs and expectations of the user). By its very nature, a DSL is already centered around the problem domain, but this does not automatically imply that it will be centered around the user's expectations and specific needs. One of the consequences of 'user-centered' design choice is that the language is designed not by focusing on what makes sense from a technology point of view, but by focusing on the user. How does the user express himself and how can we abstract from this and create a language that is both easy to use and understandable, but also provides structure?

To guide and evaluate the DSL design based on the user, I used a 'cognitive dimensions' framework [20], which gives a number of dimensions on which psychological and HCI aspects can be measured and evaluated. I will relate the different design decisions to this framework and use this to guide the tradeoffs made. An overview of this framework is presented in table 7.1.

As game designers should be able to read, and potentially, write games using the language it is very important that it is rather intuitive. Most game-designers do not consider themselves programmers, so they shy away from anything that looks like programming, however, they do feel they can use some simple visual environment or language to facilitate their needs.

Therefore the DSL should be rather visual in nature, or simple to encapsulate in an easy to use editor. Another thing that is important is to be as close to the users way of thinking as possible ('closeness of mapping'), the language should thus be rather declarative in nature.

The approach I took to developing the language was based upon the user interface that was presented to the user and how he would like to model game design aspects (top-down). This led to a number of iterations on the user interface and consequently the underlying language that supported the interface.

The top-down approach is complemented by taking a bottom-up approach where I take a look at the available technology to see what is possible. The technological possibilities provide a frame-of-reference and constraints for the DSL development. However, development was not *led* by available technology, but by user needs.

As discussed in section 5.1.1 there are many different components and component types that make up a game. Game design aspects are all components that have to deal with all the other components. Game play elements need to be specified and programmed, 3d models should be incorporated in the game, sound should be added, etc. Because of this tight integration it is important to look at how the DSL will work together with all these other disciplines. Since all of these components have these relations to the game-design, it would be nice if the DSL could somehow provide a common ground for the disciplines to base their work on.

7.3 Software Engineering

To promote good software engineer practices, reusability of code should be stimulated to allow developers to gain more in productivity. For this to happen, we would like to help programmers create reusable component, so that they can be used, preferably by a game-designer, to try and refine the game-design as soon as possible. Saving precious development time in the process.

To accomplish this, the focus of developers should be on providing reusable components that can be combined in different ways and build upon to create games. Attaining this focus is one of the goals of the DSL, and it should concentrate on facilitating programmers to implement these reusable components, instead of ad-hoc code which is often hard to reuse in a (slightly) different situation.

In section 5.2.1 I discussed the need for an extendible language, which allows the users of the language to add new concepts in a controlled way. An important requirement for the DSL is that it not only provides a language, but also a framework for extending that language to fit the particular needs of the user.

As the language will need to work based on existing technology (Cannibal Engine), the concepts it uses will have to be in-line with the concepts of this technology, the language should therefore fit the technology or lead to changes to the technology that make sense when considered in isolation. However, when the needs of a user and the constraints of the technology are in conflict, the needs of the user should get priority.

This requirement is strengthened by the fact that developers should still be able to use the core technology (Cannibal Engine) without the need for the DSL. The DSL should be an addition to the engine, not the only way to use it: developers should feel free to adopt the DSL (or not). Therefore, components that are created by programmers should be created the way they are used to, giving them full freedom and development power.

The components that are created by programmers should also be picked up by the DSL automatically, even if the programmers are not aware of the DSL. The framework provided by the underlying technology should thus allow programmers to work with it normally. The DSL should then base its language elements on whatever is created by programmers based on that technology.

A consequence of having to deal with existing technology, and the need to integrate with that technology, is that existing language prototyping tools (section 2.5) do not support the needs of this DSL project. For one the system should be visual, which eliminates a lot of the tools for textual DSL development. And second the system should fit into existing and future editing technology of the Cannibal Game Development Platform, which has strict technical requirements, which eliminate visual language prototyping tools like Microsoft DSL Tools. Because of this I chose to implement the language using the manual approach, which gives a lot more flexibility and control, but also means a lot more work.

Cognitive Dimensions	*Explanation*
Abstraction Gradient	What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
Closeness of mapping	What programming games need to be learned?
Consistency	When some of the language has been learnt, how much of the rest can be inferred?
Diffuseness	How many symbols or graphic entities are required to express a meaning?
Error-proneness	Does the design of the notation induce careless mistakes?
Hard mental operations	Are there places where the user needs to resort to fingers or pencilled annotation to keep track of whats happening?
Hidden dependencies	Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
Premature commitment	Do programmers have to make decisions before they have the information they need?
Progressive evaluation	Can a partially-complete program be executed to obtain feedback on How am I doing?
Role-expressiveness	Can the reader see how each component of a program relates to the whole?
Secondary notation	Can programmers use layout, colour, or other cues to convey extra meaning, above and beyond the official semantics of the language?
Viscosity	How much effort is required to perform a single change?
Visibility	Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

Table 7.1: The ’cognitive dimensions’ framework, taken directly from [20]

Chapter 8

Confidential

Large parts of the original content have been removed because of intellectual property protection.

Inquiries about this content can be directed at Cannibal Game Studios:

Cannibal Game Studios
Belvèdérebos 85
2725 AB Zoetermeer
The Netherlands

`info@cannibalgamestudios.com`

Chapter 9

Evaluation

Now that we have defined a DSL to model the four main aspects of game design, it is time to evaluate the DSL. Within the context of the DSL the different disciplines will be able to work together towards a complete game, while being able to reference to the same concepts and discuss the game using common terminology.

Programmers are steered towards providing reusable components that extend the language and allow game designers to define game flow. This not only allows game designers to express themselves directly into the final product, it also helps programmers create small, self-sustaining components that can be reused in different contexts.

9.1 Cognitive Framework

Here we will review the language based on the different dimensions given by the 'cognitive dimensions' framework discussed in section 7.2. Thinking about the language in terms of these dimensions allows me to evaluate how well the language is designed and identify possible areas of improvement. I will go through each of the dimensions defined in table 7.1 and evaluate the language based on each individual dimension.

Abstraction Gradient

Some elements of the DSL allow for more abstraction than others. For example, the object definitions provide abstraction by allowing new game objects to be constructed from previously defined game objects. Through this mechanism, DSL code can be reused in different contexts as encapsulated by the user.

For states, abstraction has been provided in the form of super-states and sub-states. Specifying game-flow itself has no real abstraction mechanism, since the language uses only very small graphs, although it might be interesting to see whether abstraction mechanisms might add value. Before I can decide on this, it will first be necessary to gain more detailed user feedback.

Closeness of mapping

It is difficult to establish whether the visual languages presented here are actually a lot closer to the mental models of game designers than traditional approaches. However, we can safely claim that a lot less 'programming games' [20] have to be learned when it comes to specifying game-flow and objects. There is no textual programming language to learn or specific implementation patterns that need to be utilized for working with these concepts.

Since most of the language elements are modeled according to the way of thinking of game designers, the closeness of mapping is quite well. Gameflow can be specified just the way game designers think.

Programmers may have to start using a slightly different approach than they are used to, to accomplish their code to work optimally with the DSL. We will take a look at how programmers received an early implementation of some DSL concepts in section 9.2.

Consistency

Consistency is hard to quantify, but I can make several claims as to the consistency of language in general. In my case different paradigms are implemented, which does not improve the consistency or 'smallness' of the DSL. However, most concepts are based on the same graph drawing system. This graph drawing system only has three basic elements that the user must get familiar with. Many elements also re-occur in different areas for specifying slightly different things.

To provide users with an identifier for the different language elements, all language elements have a certain specific icon within the IDE and visual language. By using this icon (with some overlays to indicate actions) throughout the interface the recognizability and consistency of the language is improved.

Diffuseness

Many of the elements of the DSL may be specified using small, self-contained graphs, but some graphs may get large, which could potentially crowd the user with information. Abstraction mechanisms have been implemented in the areas where this is most likely.

Per paradigm or concept there are only a very small number of different entities that can be drawn, which provides the user with a small terminology, allowing the different elements to be learned quickly.

Error-proneness

Because the model underlying the DSL does not allow for any syntactical errors to occur, it will not even be possible to create syntactical errors; these errors can simply not be constructed. However, like in any other language, two different semantic problems can still occur. The user might just forget to do something he meant to, for example forget to connect an event to an action or operator. These problems can usually be fixed quite easily by inspecting the corresponding graph(s).

Another error that can occur presents a bigger problem: when the user meant to do one thing and fails to specify it correctly, or when the meaning of the user is wrong. These are the common programming mistakes made that can be resolved by debugging the code. In the case of the IDE several simple debugging tools have been provided to help remedy these issues.

Hard mental operations

Since the language elements work together in such small combinations it will not occur very often that the user has to perform hard mental operations to understand anything written in the DSL. Sometimes graphs may get bigger, so that better 'secondary notation' may help to improve the understandability.

To further help the user avoid hard mental operations, several contextual hints are provided to help the user build and understand the graphs. In general the hard mental operations are also avoided by the relatively good 'closeness of mapping'.

Hidden dependencies

The DSL does a good job at specifying dependencies explicitly, most dependencies between the different elements are drawn as arrows to represent connectivity or data flow. However, some dependencies between elements belonging to different paradigms or different parts of the DSL are not always visuzlized very clearly. This may be improved upon in the future by improving the IDE, so that the browsing of software written in the DSL may indicate these dependencies by providing the proper context.

Premature commitment

Most of the elements in the DSL can just be used without the need to specify anything beforehand. However, certain kinds of elements may require some arguments that needs to be specified before creating it. For example, some extensions written by programmers may request certain parameter before allowing anything to be constructed.

I do feel this is not such a big issue as the values requested are usually very limited in number and really make sense for the designer to specify at that point.

Progressive evaluation

During development with the DSL, the user can constantly review the effects of his actions. There is virtually no state the model can be in, in which the user cannot execute the current active game object. Several game objects can also be combined into several testing objects to test how they interoperate.

It is ensured that every element in the model is valid in order for it to be present. Some combinations of elements just do not work or do not do anything when they are not well specified, however, this is not harmful to the execution of the model. As it can still be executed, ignoring these problems.

Most of the elements of a language can even be executed in real-time while editing. For example adding and removing behavior while the object is active. A game object can be told to react to certain events while instances are present, changing behavior of the object while it is being executed.

Role-expressiveness

Since many of the applications of the DSL are represnted using rather small graphs, the roles that the elements of those graphs play is rather clear. This, combined with the low count of different element types, makes sure that the role of language elements is clear.

The correlation between the different languages and language elements might not always be clear to the user. This may be due to poor user interface design on behalf of the author and some improvements could potentially be made to the IDE or the language to strengthen the relation between the different elements and paradigms of the DSL.

Secondary notation

Although the IDE does not support secondary notation at this moment, the language has virtually no restrictions on layout of different elements or their specific color. These could be used to provide further grouping of the different elements to improve upon crowded graphs. Notes or comments may also be attached to different elements by providing meta-data fields that contain this information.

Viscosity

Several improvements can be made when it comes to the viscosity of the language. Some things are a rather large amount of work to change by hand. For instance when a certain element has to be replaced by another, it first has to be removed (and its connections get lost in the process), the new element has to be inserted, and then all connections have to be redrawn. For elements with a lot of connections this is rather tedious.

Visibility

Also in the area of visibility some things can be improved, while two different game objects may be compared side-by-side, it is currently not possible to see the same object in different configurations or compare certain specific parts of a game object with another.

9.2 User Tests

Two main paradigms of the DSL have already been incorporated in the Cannibal Engine prior to it being released for the Games Project at TU Delft (see sections 4.2.2 and 5.2). This allows me to already provide a short evaluation of how these elements were used by programmers without being explicitly attended to it. Because the programmers were not attended to the systems in an explicit way the natural tendency to start using such an approach can be evaluated.

Events

As explained in section 5.2.2 an event system has been implemented quite early in the thesis project. The system available for the games project allowed for events and operators to be implemented and used. Tying them together and to actions requires a more manual approach than drawing graphs, where programmers have to specify event handlers and call actions, etc.

Several groups of the Games Project have implemented their own events and operations, but only on a very small scale. However, the system was used throughout the code for handling events by tying them to different actions. Operators were used as well, but usually only the ones that were provided and were quite obvious to use.

Whenever programmers asked how to accomplish certain things that could best be modeled using the event system, the system was briefly explained to them. It is quite interesting to see that the two groups to which we explained the system the most also started to find applications of the system on their own. They implemented more operators and started to use them more frequently, some of these operators are now even part of the core package provided by the engine.

So in general, groups that got to know the system a little bit better started making more use of it. This makes me believe the system in essence is very powerful, but not that obvious to programmers at first. This might be due to the fact that they are not used to write code in a declarative and event-oriented fashion.

Objects and Attachables

The second element that was already present in the Cannibal Engine was a predessecor of the DSL object specification paradigm. The scenegraph implemented for the Games Project consists out of general *nodes* that only designate a location and orientation in the 3d world and several *attachables* adding semantic value to that node. The functionality provided by the Cannibal Engine has already been implemented using that approach and users were forced to use the approach when they required that functionality. But did they use such an approach themselves?

As can be seen from the source trees of the different groups, almost all groups used the system consistently to define their own game objects and shared functionality between them. A difference to the paradigm of the DSL is that whenever a certain functionality was only applicable to one group of game objects, classic object-oriented inheritance would be used to share the functionality.

In some cases, where functionality was to be shared between very different types of game objects, the attachables were used to provide this behavior. This is more inline with the approach implemented in the DSL. These attachables were mainly created for very common tasks like collision detection and automatic camera/object controlling, etc. These were also programmed so that they could be reused in many situations and different games. From these results, it seems that the approach of attachables, and most likely the object specification paradigm in general, can easily be adopted by the programmers on the team.

During the development of the DSL the several languages and the IDE have been shown to several artists. Artists are not necessarily game designers, but they do have a liking for games and no real technical background. Using their comments the user interface has been changed several times and has become what it currently is. In general most artists understood the system pretty well after a brief explanation and they were able to create small examples with it.

However, no real formal usability testing has been performed on the actual target groups: game designers and game play programmers. The feedback that was received was all gathered during very informal talks where early prototypes of the system were demonstrated and shown.

9.3 Future Work

Work in the area of the DSL is not complete, so there are several things that are left to future work that did not fit into the scope of this project.

The current IDE is not a polished tool like users would expect and it has not been integrated with existing toolsets and the collaboration platform. Integrating and improving the IDE to allow for a better editing experience is one of the essential things that must be done before the DSL can be used as part of a product like the Cannibal Game Development Platform.

The system has been tested using small scale tests only. Testing the language in a real-life setting, not in student projects and proof-of-concepts, might prove useful to uncover potential problem areas or areas of improvement. Results of this tests may be used to either decide to research and implement potential language extensions. These can then be fed back to the users to test whether they indeed provide an improvement.

Since performance is always an issue when it comes to game development, it might be wise to research what the performance penalty is for using the language. Throughout the language and the support framework care has been taken to not adversely effect performance, however, in some areas performance improvements might still be gained.

One thing that might significantly improve performance is compiling the DSL model to an implementation in C# or general .NET code (MSIL) [14], which can then be used without requiring the DSL model to be running as well.

Many examples used to test and base the language on where extracted from real-life code, where people have programmed game design aspects in a GPL (using the event language as a library). These were translated into the DSL to see whether it would be able to recreate that particular game design aspect and to learn from missing aspects to the language.

For the future it would be interesting to see whether automatic extraction of a DSL representation from the code would be possible, to be able to extract pieces of code that were initially written by a programmer and present these to the game designer.

Chapter 10

Conclusions

Developing a domain-specific language for computer games is quite an open assignment. In this thesis I argued (in section 5.1.1) that some parts of creating a game are already supported by great tools, while the game design aspect is not supported at all, or quite poorly. The game design aspect is currently implemented using conventional (general purpose) programming languages, not geared towards solving specific problems. Game developers are confronted with a lot of technical details when it comes to programming the game design elements. It is in this area the DSL can help by moving this work to a higher abstraction level.

There are several aspects to implementing a game design, which are all of great importance (see section 5.1.3). Providing support for these aspects can be considered as the main functional requirements for the DSL and the definition of the domain for the DSL. These aspects are:

- *The objects* that make up the game world, with specific behavior and properties.
- *The interaction* with the player, determining how he controls the game and what is given back to the user as feedback.
- *The rules* that govern the core mechanics of the game and determine what the player can and cannot do.
- *A storyline* is not present in all games, but in some genres and games it takes a prominent place.

To provide supports for these aspects a DSL has been designed and formally defined. This DSL consists of several different elements to help specify the different game design aspects. It thus contains elements for specifying structure of game objects, rules, interaction, and states of objects. Each of these elements work within a paradigm specifically suited for their application area.

Since the DSL is mostly visual, an IDE has been designed and implemented to allow developers to work with the language. However, users of the language are not forced to use that specific IDE or visual representation. The DSL has been implemented in such a way that the actual game model has been separated from the concrete syntax of the language, allowing different representation of the same model. In fact, when the game is executing, no other representation than the executed game is present.

To facilitate the implementation of the DSL, several aspects of the Cannibal Engine have been changed to provide a relevant underlying support framework. Through this framework programmers can extend the DSL to fit the particular needs of the game studio and the game under development. Extensions are created in such a way that they can easily be reused for other projects or in different contexts.

During the development of the DSL for the computer games domain it has become clear that using a DSL not only brings advantages, but also disadvantages. It takes a lot of time for a DSL to be developed and to keep every potential issue in mind (e.g. performance and usability). However, I do feel that the advantages the DSL can bring are worth the troubles of designing and implementing it.

Although the DSL here is not ready to be shipped as a final product yet, it provides a proof of concept and the groundworks for a new and interesting tool specifically for game designers. The DSL helps game designers express themselves more directly into the final product and provides a common language for different disciplines to discuss the game.

Before the DSL can actually be used as a product, several things will have to be researched or improved. First and foremost, the IDE will have to be improved and integrated with the existing toolset and the collaboration platform.

Of course potential language extensions should be considered and researched. Bigger scale, real-life tests can prove valuable in gaining knowledge about these extensions and the workings of the DSL. The tests can also provide valuable information as to the performance of the framework. Where needed this performance should be improved to allow high-performance (or triple-A) games to be created.

For the future it would be interesting to see whether it would be possible to automatically extract existing game code and provide a matching DSL model for this. This would lower the barrier for game developers to start using the DSL and may prove useful in reusing already existing solutions at game studios.

My recommendation to Cannibal Game Studios as a first step is to turn this work into a practical tool, integrated with the existing toolset, to allow users to experiment with the DSL and use it in real-life settings. This will not only provide the game industry with this tool in a relatively short amount of time, it will also provide valuable user feedback that can be used to further improve the DSL.

Bibliography

[1] Multiple Anonymous authors. Duke nukem forever. http://en.wikipedia.org/wiki/Duke_Nukem_Forever.

[2] Multiple Anonymous Authors. Frag website. http://haskell.org/haskellwiki/Frag.

[3] Multiple Anonymous Authors. Upgear website. http://www.upgear.nl/.

[4] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[5] G. Berry. The esterel v5 language primer - version 5.10, release 2.0.

[6] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[7] M. H. Cheong. Functional programming and 3d games. Master's thesis, The university of New South Wales, 2005.

[8] Web3D Consortium. X3d website. http://www.web3d.org/.

[9] Web3D Consortium. Extensible 3d (x3d). Technical report, Web3D Consortium, 2004.

[10] XML Game Consortium. Gamexml website. http://www.gamexml.org/.

[11] Intentional Software Corporation. Intentional software website. http://intentsoft.com/.

[12] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[13] A. Van Deursen and E. Visser. Model-driven software evolution, interaction and co-evolution of models and code, 2006.

[14] ECMA, December 2001. Standard ECMA-335: Common Language Infrastructure (CLI).

[15] C. Elliott. Modeling interactive 3d and multimedia animation with an embedded language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 285–296, 1997.

[16] C. Elliott. Declarative event-oriented programming. In *PPDP '00: Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 56–67, New York, NY, USA, 2000. ACM Press.

[17] Digital Extremes Epic Games. Unreal, 1998. Computer Game.

[18] UML Revision Task Force. Unified modeling language: Superstructure. Technical report, Object Management Group, 2004. http://http://www.omg.org/docs/formal/05-07-04.pdf.

[19] S. Gourlay. Tacit knowledge, tacit knowing or behaving? In *OKLC 2002: Proceedings of the 3th European conference on Organizational Knowledge, Learning and Capabilities*, pages 269–287. -, 2002.

[20] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[21] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[22] The Khronos Group. Collada website. http://www.khronos.org/collada/.

[23] The Khronos Group. The khronos group website. http://www.khronos.org/.

[24] The Khronos Group. Collada digital asset schema release 1.4.1, specification. Technical report, The Khronos Group, 2006.

[25] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[26] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[27] C. S. Horstmann and G. Cornell. *Core Java 2: Volume I, Fundamentals, Sixth Edition*, chapter 8, pages 277–334. Pearson Education, 2002.

[28] Appaloosa Interactive. South park, 1999. Computer Game.

[29] A. B. Sutman J. M. Schementi. Video game language. Master's thesis, Worcester Polytechnic Institute, 2005.

[30] R. M. Herndon Jr. and V. A. Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14(6):803–809, 1988.

[31] B. Keller. Xna studio: Introduction to xna, 2006.

[32] S. Kent. Model driven engineering. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 286–298, London, UK, 2002. Springer-Verlag.

[33] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 542–552, Washington, DC, USA, 1996. IEEE Computer Society.

[34] D. Kushner. *Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture*. Random House Trade Paperbacks, 2004.

[35] B. Leuf and W. Cunningham. *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley Professional, April 2001.

[36] Microsoft. Microsoft domain-specific language tools. http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx.

[37] D. A. Norman and S. W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 1986.

[38] Zillions of Games. Zillions language reference - version 2. Technical report, Zillions of Games, 2003.

[39] M. Overmars. Game maker website. http://www.gamemaker.nl/.

[40] M. Overmars. Game design in education. Technical Report UU-CS-2004-056, Institute of Information and Computing Sciences, Utrecht University, 2004.

[41] M. Overmars and J. Habgood. *The game maker's apprentice: game development for beginners*. Berkeley, CA: Apress, 2006.

[42] B. Rector and J. Newcomer. *Win32 Programming*, chapter 1 and 2, pages 1–102. Addison-Wesley, 1997.

[43] A. Rollings and E. Adams. *On Game Design*. New Riders, 2003.

[44] W. C. Savenye and R. S. Robinson. *Qualitative research issues and methods: an introduction for educational technologists*, pages 1045–1072. Lawrence Erlbaum Associates, 2004.

[45] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 451–464, New York, NY, USA, 2006. ACM Press.

[46] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang. Organization domain modeling (odm) guidebook version 2.0. Technical report, Synquiry Technologies, Inc, 1996.

[47] G. Smolka. An Oz primer. Technical report, German Research Center for Artificial Intelligence (DFKI), 1995.

[48] Sony. Playstation 3 website. http://www.khronos.org/.

[49] Cannibal Game Studios. Cannibal website. http://www.cannibalgamestudios.com/.

[50] Cannibal Game Studios. Cannibal game development platform api, 2004-2007. Internal Document.

[51] D. Suttles, R. Chen, M. Paugh, and A. Mandic. Metanode organized prototype hierarchy specification (morph) - version 0.9b. Technical report, Magnetar Games, 2006. http://www.gamexml.org/Specs/MORPH/.

[52] T. Sweeney. Unrealscript language reference. http://unreal.epicgames.com/UnrealScript.htm.

[53] T. Sweeney. The next mainstream programming language: A game developers perspective, 2006.

[54] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, 2000.

[55] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370, London, UK, 2001. Springer-Verlag.

[56] A. van Deursen and P. Klint. Little languages: little maintenance. *Journal of Software Maintenance*, 10(2):75–92, 1998.

[57] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[58] E. Visser. Stratego/xt. http://www.program-transformation.org/Stratego/WebHome.

[59] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[60] Z. Ye. Genres as a tool for understanding and analyzing user experience in games. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 773–774, New York, NY, USA, 2004. ACM Press.

Appendix A

Glossary

Abstract Syntax Tree The term abstract syntax tree refers to a tree, where the internal nodes represented operators and the leaf node represent operands. They are used to represent computer software in an abstract way (non-language and non-platform specific).

Assets Game assets are the ”things” that go into a game. Some examples of assets are artwork (including textures and 3D models), sound effects and music, text, dialogue and anything else that is presented to the user. Sometimes the terms content or objects are used interchangeably with the term assets.

Content pipeline The content pipeline is the steps that needs to be taken to get assets into the game. The supporting tools for these steps are usually included as being part of the content pipeline.

DCC Tools Digital Content Creation tools: tools used to create different types of digital assets used in computer games.

Declarative Programming A programming paradigm in which the developer focuses more on ’what’ should be done, than on ’how’ it should be done (sequence of commands and state changes). (This is often contrasted against imperative programming.)

Domain Specific Language (DSL) A language that is geared towards solving a problems in a specific domain, or sometimes specific problems in a domain.

Event-Driven Programming Event-driven programming is a programming paradigm in which the developer can structure the program by specifying the reaction of the program to external events or actions.

First playable A very early prototype of the game that contains the most important gameplay elements. This allows developers to test concepts and provides a basis to build upon.

Framerate / FPS The number of updates/draws per second that the game is performing. FPS is an abbreviation for ’frames per second’.

Game Design The complete package which determine the game. This contains game rules, object definitions, interaction specification, etc.

Game Designer The member of the game development team who designs the rules of the game, game play aspects, characters, story, etc.

General Purpose Language (GPL) A language created to solve problems in many different domains.

Imperative Programming A programming paradigm in which the developer gives a sequence of commands to be executed, changing the state of the system. (This is often contrasted against declarative programming.)

Leads The more experienced members of a game development team are often called leads. For example, the Lead Artists is usually the most experienced artist in the studio (apart maybe from the Art or Creative Director).

Level designer Very much like a Game Designer, only this member of the development team designs specific levels and interactions within that level.

Look & Feel The way something looks and is experienced by the user, for example: the look & feel of Windows XP is different than that of many Linux environments.

Material The material of an object determines the way the object looks by defining a texture, reaction to lights and other special properties.

Next-gen The next generation of, in this case, computer games and computer game technology. This is a typical term used for very new hardware, software and consumer demands.

Paradigm A way of looking at the world; a particular view of the world in reference to a particular discipline. It refers to a number of tacit assumptions and believes shared by certain members of that discipline. It is the conceptual model underlying the theories and practice of a scientific subject.

Particle effects Special effects used to simulate things that consists of many small particles, like smoke, fire, snow and explosions.

Physics Simulation of physical laws and constraints, usually, but not necessarily, based on the real laws of physics.

Protagonists The most important characters in a story. This terms is used more widely in movies.

Rigid bodies Non-deformable bodies that are used to simulate the physical behavior of objects in games.

Scenegraph A scenegraph is the dominant data-structure for storing the logical composition of 3d scenes. Objects are represented by nodes in the scenegraph. Usually the scenegraph is actually a tree, where you start at the root, and children are connected. Children are affected by transformations on the parent nodes.

Shader A piece of software that runs on the grapical processing unit in stead of the central processing unit of the pc.

Tacit Knowledge Something that is understood or implied without being able to explicitly state it.

Texture An image that is projected onto an object to provide it with patterns and colors.

Appendix B

ViGL - Tic Tac Toe

```
<vigl resolution="300x300">

<code location="begin">
$TURN = true
class GameObject
  def inside?(mouseState)
    if(mouseState.button==Mouse::BUTTON_LEFT and
      mouseState.x >= @renderable_object.rectangle_shape.point.x
        and
      mouseState.y >= @renderable_object.rectangle_shape.point.y
        and
      mouseState.x < @renderable_object.rectangle_shape.point.x +
               @renderable_object.rectangle_shape.dimensions.width
        and
      mouseState.y < @renderable_object.rectangle_shape.point.y +
               @renderable_object.rectangle_shape.dimensions.height
    )
      true
    else
      false
    end
  end
end
</code>


<objectdef name="Square">
  <shape><square length="100" />
         <graphics border="#000000" color="#ffffff"/>
  </shape>
  <actions>
    <method action="onMouseDown(state)"><code>
      if(inside? state
         and
         @renderable_object.renderable_properties.fill ==
         Color::WHITE
      )
        @renderable_object.renderable_properties.fill =
          if $TURN
            Color::RED
          else
            Color::BLUE
          end
        $TURN = !$TURN
      end
    </code></method>
  </actions>
</objectdef>

<world>
  <object parent="Square"><shape><square point="0,0" /></shape></object>
  <object parent="Square"><shape><square point="100,0" /></shape></object>
  <object parent="Square"><shape><square point="200,0" /></shape></object>
  <object parent="Square"><shape><square point="0,100" /></shape></object>
  <object parent="Square"><shape><square point="100,100" /></shape></object>
  <object parent="Square"><shape><square point="200,100" /></shape></object>
  <object parent="Square"><shape><square point="0,200" /></shape></object>
  <object parent="Square"><shape><square point="100,200" /></shape></object>
  <object parent="Square"><shape><square point="200,200" /></shape></object>
</world>

</vigl>
```

Appendix C

Zillions of Games - Tic Tac Toe

```
(game
  (title "Tic-Tac-Toe")
  (description "One side takes X’s and the other side takes O’s.
   Players alternate placing their marks on open spots.
   The object is to get three of your marks in a row horizontally,
   vertically, or diagonally. If neither side accomplishes this,
   it’s a cat’s game (a draw).")
  (history "Tic-Tac-Toe was an old adaptation of Three Men’s Morris    to
   situations where there were no available pieces. You can draw or
   carve marks and they are never moved. It is played all over the
   world under various names, such as ’Noughts and Crosses’ in
   England.")
  (strategy "With perfect play, Tic-Tac-Toe is a draw. Against less
   than perfect opponents it’s an advantage to go first, as having an
   extra mark on the board never hurts your position. The center is
   the key square as 4 possible wins go through it. The corners are
   next best as 3 wins go through each of them. The remaining
   squares are least valuable, as only 2 wins go through them.
   Try to get in positions where you can ‘trap‘ your opponent by
   threatening two 3-in-a-rows simultaneously with a single move. To
   be a good player, you must not only know how to draw as the second
   player, you must also be able to takes advantage of bad play.")

  (players X O)
  (turn-order X O)
  (board
    (image "images\TicTacToe\TTTbrd.bmp")
    (grid
      (start-rectangle 16 16 112 112) ; top-left position
      (dimensions ;3x3
        ("top-/middle-/bottom-" (0 112)) ; rows
        ("left/middle/right" (112 0))) ; columns
      (directions (n -1 0) (e 0 1) (nw -1 -1) (ne -1 1))
      )
    )
    (piece
      (name man)
      (help "Man: drops on any empty square")
      (image X "images\TicTacToe\TTTX.bmp"
       O "images\TicTacToe\TTTO.bmp")
      (drops ((verify empty?) add))
    )
    (board-setup
      (X (man off 5))
      (O (man off 5))
    )

    (draw-condition (X O) stalemated)
    (win-condition (X O)
      (or (relative-config man n man n man)
          (relative-config man e man e man)
          (relative-config man ne man ne man)
          (relative-config man nw man nw man)
      )
    )
  )
```

Appendix D

Interview

D.1 Company Info

1. What is the current total number of employees of the company?
2. Team structure:
    a) What are the functions in the company?
    b) What do they do exactly and why?
    c) How do these functions relate to each other?
    d) Per function how many employees have that function?
3. Activities:
    a) In what type of activities is the company engaged and what is the core activity of the company.
    b) How many activities are done simultaniously.
    c) What products have been produced in the past.

D.2 Collaboration

1. What collaboration tools are used in the company and do they meet the demands?
    a) Communication systems like email and Wiki.
    b) Version control system like svn and cvs.
    c) Project-management systems.
2. Are there any points to be improved regarding the collaboration / communication.

D.3 Work Process

1. Disciplines:
    a) What are the specific disciplines in the development team? (engine, AI, graphics, art, gameplay, etc)
    b) What type of assets are needed and created by these people? (models, textures, concept art, etc)
2. How well can groups from different disciplines work independently? Which tasks require a joint effort of different groups.
3. What steps do you follow during development of a game (ideas and requirements, design, implementation, publishing).
4. How do these steps relate to each other concerning time spend.
    a) What takes up the most time?
    b) Where do *you* feel time is 'lost'?
5. How does content get into the game?
    a) Who does it?
    b) How is this done? (custom editors, etc)
6. What are the bottlenecks during the development? What would be possible solutions?
7. Software and tools:
    a) What tools and software packages are used by the groups (version control, communication, project management, bug/task tracking)?
    b) How do these tools work together?
    c) Why do you use *these* tools?
    d) What are the advantages and disadvantages of these tools?
8. Is there anything you want to add concerning the work process of your company?

D.4 Engine

Ask these questions to every individual/function.

1. What engines do you have experience with?
    a) Pro's of the engine.
    b) Con's of the engine.
2. From your point of view:

   a) What would be the ideal way of working with an engine.
   b) What are important features of an engine? (multi-platform, cool graphical features, good workflow)
   c) What do you need the engine to do for you?
   d) What do you want to do yourself?
   e) What parts of the engine would you like to be able to customize or add functionality to?
   f) How would you like to have support? (e-mail, phone, representative, on-line (wiki/forum))

D.5 Game Design

1. Could you give me a complete specification of the game pacman/tic-tac-toe (in game design terms)?
2. How do you specify the rules of a game and the game design?
3. How does a game idea end up in the final game, what are the steps involved?
4. How do you determine and communicate the look & feel of the game? (somehow formalized or informal documents)?
5. Cooperation on game design:
   a) Who decides what?
   b) How would you work with others on the design?
6. What are common things you change to the game/design?
7. Do you take into account the technical limitations and implications when you design the game? To what extent?

D.6 World creation

1. How would you describe the creation of the game world.
2. Cooperation
   a) What is the function name of the person who creates the game world?
   b) Is creating the game world a single person task or do people work together?
3. Process
   a) What steps are involved in creating the game world?
   b) How much time does each step take?
4. Which tools are used in creating the game world?

a) Pro’s of the tools.

b) Con’s of the tools.