Scripting Game AI: An Alternative Approach using Embedded Languages

Andrew Calleja
Computer Science Department
University of Malta
Msida, Malta
acal010@um.edu.mt

Gordon Pace
Computer Science Department
University of Malta
Msida, Malta
gordon.pace@um.edu.mt

ABSTRACT

Scripting is often used in games to enable customisation of the behaviour of game entities. In this paper we look at the different approaches taken to introduce scripting in games and identify the desirable features of a game scripting language. We then present an approach based on the use of embedded languages where a scripting language is implemented within a general purpose programming language. The approach is compared and contrasted to the other approaches.

1. INTRODUCTION

In games, scripts are often used to program what entities in fictitious worlds say or do and also how they react or behave when a player interacts with them. They are thus essentially programs relating to a particular character or object which are executed in order to give the player a better immersive experience while playing.

There are a number of commonly used methods by means of which scripting may be introduced within a game system and they differ in their attributes. Game developers often have to select the method which best suits their needs. The attributes are based on a number of basic factors which a particular method might lack or excel in. Since a programming language is often used to script the AI, the first set of factors are related to the former itself. Is the language easy to implement, tailor, update and integrate? The next set of factors relate to the method's target audience. Is the method going to be used by the game programmers themselves for the specific purpose of creating the game's required AI or does it also require to be simple enough for non-programmers such as story-writers to be able to write the required plot? Finally, there are a number of features which are ideal to have within the method since they allow for a better experience whilst making use of the method itself. Is there a separation between the game logic and the scripts themselves such that changing the script does not imply a recompilation of the entire game? Does the method require scripts to be interpreted or compiled either at game compile-time or even in real-time? If we wish to allow third party programmers, such as fans and any company which might expand upon the game, to write extra features or extensions to the game once it has been completed, does the method we employ allow this? All these factors play a pivotal role in the selection of the required scripting method.

In this position paper we will discuss the three most common scripting methods in light of the above common attributes. After comparing these approaches we will propose a fourth approach which bases itself on the strong-points of the other approaches and which we believe provides an adequate alternative. This approach makes use of the technique of language embedding where a language, in our case the required scripting language, is implemented via embedding within another language which acts as host. As case studies we will make use of our approach to implement the scripting languages for two games of differing genres and use these languages to script the games' AI. Moreover, it is possible to make use of the host language as a meta-language over the embedded scripting language allowing for on-the-fly script-generation based upon the game state. In this manner, we are able to evaluate our approach qualitatively based upon the common attributes which in turn allows us to compare and contrast it to the other approaches.

2. GAME-SCRIPTING APPROACHES

At a very high level a game's scripting system involves the interactions of three modules: the game engine, the game API and the scripting engine. At the lowest level we have the game engine which lies at the core of any game and is responsible for managing the game's logic as dictated by the game's design. It also interacts with and coordinates various other modules such as those dealing with the processing of input/output and with the loading of any content such as graphics, sounds and scripts [12]. Another module with which the game engine interacts is the game API. The game API is used to make calls to the game engine which the latter then uses to query or update the game's state. The API thus acts as a gateway with which the game engine interacts safely with the outside world [12]. Another module, known as the scripting engine, is able to communicate with the game engine via the API. This module is the one which handles scripts themselves and often acts as a form of interpreter over scripts. When a script needs to be run the scripting engine interprets it and extracts from it a number of API commands which are then carried out [12]. Implementing this system can be done in various ways which vary both in how closely coupled the three modules are and the type of language used for scripting itself amongst other factors.

2.1 Game Engine Integration

An initial attempt at creating a game scripting system is one where the latter is integrated directly within the game engine [12]. Since the game engine and the scripting engine are so closely coupled, the language used is often the language with

which the game engine was written with. Thus the scripts themselves take the form of API calls and are usually hard-coded within the game engine itself [12]. This approach brings with it the advantage that it allows the programmer who wrote the game engine to also write the scripting part for the game. Also, since there is no interpretation required, just calls to the API, this is by far the fastest approach when compared to other approaches which we shall discuss shortly.

Unfortunately, since the scripts themselves are essentially modules within the game engine, each time these need to be updated or modified, the game itself must be recompiled. This should not be the case as the scripts should essentially be content, just like any sound files or 3D meshes, which are loaded by the game only when required [12]. In many games the author of the content is often not a technical person. More often then not they are writers who contribute artistically to the game in the form of lore or plots by writing script dialogue and actions. It is highly improbable that they know or understand the language with which the game engine was written in order to write the required scripts for the game. Due to this the game programmer must translate as closely as possible the writer's script and if this is not done well or without feedback from the writer themselves it could make or break the game [4]. Finally, the game is a closed world and apart from taking it apart and rewriting the modules which encode the scripts themselves it has no way of allowing the inclusion of any third party code which changes/adds (known as mods or add-ons) or facilitates (known as macros) certain aspects of the game [12].

2.2 General-Purpose Language Scripting

Another way scripting may be introduced within a game is by adding a scripting engine module as a separate module sitting apart from the game engine. This module receives as input a script written in a general-purpose language which is often similar to C++ or Java. Acting as a lightweight virtual machine it then interprets or compiles the script into the required API calls which are then run by the game engine [12].

Many games make use of this solution as it solves most of the problems found in the approach described earlier. The first obvious advantage this method provides is that it imposes a separation between the scripting engine and the scripts themselves. If changes need to be made to the scripts this can be done at ease without having to edit the game engine itself. This also removes the need for constant compilation whenever a script needs to be changed [12]. Another advantage is that the general-purpose language used itself is often similar to the one used to program the engine. This allows the programmers of the engine to be more familiar with the language used and it enables them to write the scripts themselves if this is required. This approach also has the final benefit of allowing the writing of mods and macros with relative ease [12]. What is required is only one's familiarization with the language used for scripting. Often the language selected is a popular off-the-shelf scripting language such as Python, Lua and Ruby [6]. For example, Python has be used to script the turn-based strategy (TBS) game Civilizations IV and the massively online role-playing game (MMORPG) EVE Online.

As an example of general-purpose language scripting we present a small example in Python which involves the player taking the role of a hero in a fictional world and his or her interactions with a scripted male farmer. The farmer reacts with hostility to the hero when the latter is poor since he expects that he or she will attempt to rob him. Otherwise, he greets the hero formally by name and queries him or her how he may be of assistance:

```
import api
def script(hero) :
  if api.interacted(hero) and api.poor(hero):
    api.say("I don't want trouble")
  else: api.say("Hello " + api.name(hero) + ", can I help you?")
```

The first line imports the game's API module which acts as a wrapper around the actual game engine such that the functions `interacted`, `poor`, `say` and `name` can be used to query or modify the game state. The rest of the code shows how we can use a Python function (by making use of the keyword `def`) and python's conditional `if` statement in order to write the required script.

The main disadvantage of this approach is that developing a scripting language such as Python or Lua is not a trivial process. In fact, most users of this approach employ the use of an off-the-shelf language such as the latter since this saves the time and cost building a custom scripting language. However, these off-the-shelf variants are often over-bloated with features which might not be required by a particular game. This might cause them to be quite slow when compared to other game scripting systems. Also, they are technically inclined and are easier to understand by a programmer rather than a content designer. They thus still require the translation of the latter's scripts into actual code. Finally, integration is not straightforward to achieve and requires a good amount of work to link the scripting language to the game engine's API [12].

2.3 Proprietary Language Scripting

A third approach to scripting is by making use of a language designed specifically for the purpose of scripting a particular game. Such a language is often not designed to be generic but domain-specific to the particular game or family of games making use of the engine [12]. This approach shares many advantages found in general-purpose scripting languages in that separation is still maintained between the game engine and scripting engine. This allows the scripts to be seen as content apart from the game engine itself hence removing the need for compilation whenever a script needs to be changed. Moreover, this allows for the possibility of allowing mods and macros to be written using some form of plug-in system. Another advantage is that the language may be completely customized and this allows the possibly of making it domain-specific. If written in an appropriate manner it could allow content designers to write the scripts themselves.

There are various forms of scripting languages. Some of them are command-based and consist of simple commands. Such languages often do not include more complex constructs such as loops and are thus very hard to use to implement complex scripts such as a character's AI [12]. Varanese [12] gives an example of such a language for a generic RPG

game which is reproduced here:

```
MovePlayer 10, 20
PlayerTalk "Something is hidden in these bushes..."
PlayAnim SEARCH_BUSHES
PlayerTalk "It's the red sword!"
GetItem RED_SWORD
```

As can be noticed the language is very domain-specific and includes only commands related to the game such as moving the character to a location. Thus, these languages are thus easier to understand by non-programmers. On the other hand, other languages are almost similar in nature to general-purpose languages but restrict themselves to the game engine only in their scope. Such languages are better suited to encode AI by programmers. An example of such a language is UnrealScript [1]. The language itself is very similar to Java in nature and is object-oriented. Unfortunately this makes it less attractive to content designers. An example of UnrealScript code is the following:

```
class HelloWorld extends Mutator;
function PostBeginPlay() {
  Super.PostBeginPlay();
  Log("Hello World");
}
```

This simple example outputs "Hello World" to the game's log file. It consists of a class which extends a more abstract `Mutator` class. This latter class defines an actor in the game world. What is important to notice here is that we have a variant of Java which is restricted to the character or objects acting in the game world.

Proprietary scripting languages are often ideal since they incorporate all the advantages of general-purpose scripting languages. However, like the latter, creating a language from the ground up is often seen as outweighing the benefits. Designing a language is often not an easy task and requires a lot of work extracting what the language's needs are. Also, implementing the required tools such as compilers and interpreters takes a non-negligible amount of time even for the simplest of languages. Finally, fully optimising such tools is not an easy feat. Thus, developing such languages might only be feasible when the game engine is being planned to consist of a complete suite to be sold to and used by third party developers. In such cases it becomes more feasible to create a new language since it can act as one of the features in the game engine's selling-points.

2.4 Desirable Attributes of a Scripting Method

After examining these three approaches we can deduce the ideal attributes of a scripting method:

Language implementation: the amount of work and time required to implement the method's scripting language itself. Language implementation should not be so complex as to require substantial development-time. The scripting method employed should allow us to implement the scripting language as easily and quickly as possible.

Language updating: whether significant work is required in order to add new features to the method's language. Usually updating a language requires a lot of work identifying where a change needs to be made, studying any side-effects and then implementing the change itself. It is desired that updating the scripting language is not a major task since in case of games this could occur often during the latters' evolution.

Language tailoring: how easy it is to alter the method's language towards a particular user-base such as script-writers or AI-programmers. Tailoring a language normally requires work in determining the required user-base's needs and then altering the language accordingly. It should not be the case that tailoring a language requires enough work as to be comparable to implementing a different language each time.

Language integration: how much effort is required in order to integrate the method's scripting language within the game's implementation language in order to actually make use of. Language integration is usually a non-trivial step in the use of the scripting language at hand since appropriate wrapper functions must be created mapping to the game's API accordingly. Language integration is a necessary step in any scripting method, but which should be minimized as much as possible.

Language usability: how accessible and expressive the method's scripting language is. Usually the scripting language itself needs to strike a balance between being accessible enough for use by non-programmers in order to write content scripts and at the same time expressive enough in order to allow programmers to write AI scripts. Ideally, if this is required, both programmers and non-programmers can make use of the scripting language for their needs, even if this is often not possible.

Logic/content separation: game engine/scripts separation such that the former does not require recompilation each time the latter change. Separating the game engine and the scripts is a desired component of any scripting method as it saves time during the game-creation process by removing the need for constant game-engine recompilation.

Third-party scripting: the method's support for mods and macros. Mods and macros may form an integral part of the game itself and, if included, have often been used to give the game players a means of extending the game itself, altering it significantly or automating repeated parts of it. Having a scripting method which allows for this is often desirable for the game's longevity.

General-purpose language available: whether the power of a full general-purpose language is available. Certain languages, such as certain game-specific scripting languages, do not have this power and are often used for content-scripting only. For the sake of most AI scripting, however, a general-purpose language should be available for use in order to allow decisions to be taken within the scripts.

The methods described above excel in some attributes while performing poorly in others. In this body of work it is our interest to propose an alternative method which attempts to maximise upon their strengths. To achieve this we need a way to combine a language consisting of domain-specific constructs tied to a game with a general-purpose language which includes looping and conditional constructs. In this

way such a language can be used both by story-writers in story-line scripting and programmers in AI scripting. Finally, we wish that if we have to implement, tailor, update and integrate such a language ourselves, these processes do not require such a huge effort as to be inadequate for use during game creation.

3. DOMAIN-SPECIFIC EMBEDDED LANGUAGES

In recent years a versatile technique has been developed where a language pertaining to a particular domain, called a domain-specific language, can be embedded within a host language. Such a domain-specific embedded language is not developed from scratch as is the case with normal domain-specific languages, but rather a host language, which is usually a general-purpose language, is adapted to supply the syntax and semantics and thus act as the embedded language's compiler or interpreter.

In 1966 Landin [9] remarked that a language consists of a basic set of constructs and a number of ways in which to combine them but stressed the fact that the suitability of a language towards a domain is closely tied to the former rather than the latter. Thus, by identifying a number of basic constructs related to a domain, an appropriate domain-specific language (DSL) for the latter is formed. Landin then proceeded with building upon such a notion of a DSL by proposing domain-specific embedded languages (DSELs) as a natural step forward. To achieve this he suggested the creation of a general-purpose language which can be geared towards a particular domain by selecting a number of basic constructs. The first actual uses of language embedding were done via the functional language Lisp's [11] macro system. More recently, Hudak [7, 8] reintroduced the approach and made it popular as a viable methodology to develop programming languages for subsequent use in software development. DSLs offer the right amount of abstraction for software projects but creating a new language from scratch each time requires a considerable amount of work in terms of language design and tool development. Hudak thus suggested the use of an existing infrastructure of an existing language whose properties meet the requirements of the DSL. Using such a host language's mechanisms and tools, adapted to a particular domain it is possible to create the required DSEL with the added advantage of reducing costs, time and effort. The work required is only in adding the domain-specific functionality to the host language.

Functional languages are often selected as host languages due to their features which include pattern matching, lazy evaluation, module system, higher-order functions, strong typing, polymorphism and overloading which are useful for language embedding [8]. These combined features allow for a level of abstraction which enables the user to focus on the domain itself rather than having to consider a lot of implementation details which clutter domain-specific thoughts. Nowadays, the language of choice for embedding is Haskell since it has the features just mentioned while adding some of its own such as monads and a non-restrictive syntax. Languages related to various domains successfully embedded in this language include: images [5], hardware description [2] and business processes [10], amongst many others.

When embedding, the selected domain-related abstractions are encoded via the host language data types. These data types provide us with the syntax of our embedded language. Haskell's syntax allows us to create an embedded language which is free from annotations which are not domain-related, something which often plagues other hosting languages, resulting in a look and feel of a separate language rather than an embedded one. The semantics of our language are provided by a number of functions in Haskell itself which act as the language's compilers or interpreters. By traversing the data structures created by the domain-related data types they attribute a meaning to such structures and return the required result of interpretation.

4. EMBEDDED GAME SCRIPTS

The technique of language embedding has allowed us to suggest a new, alternative approach to game-scripting which scores adequately with respect to the basic factors mentioned earlier. In regards to ease of implementation, the embedded language approach enables us to quickly and efficiently create scripting languages without actually requiring us to develop traditional tools such as compilers and interpreters from the ground up. Moreover, altering the language in any way is also a very straightforward process. Integrating the language is also simple since we are essentially still programming in the host language. In fact, implementing the scripting language using the embedded approach means it is seamlessly integrated within the language used to program the game itself. Since we are developing a domain-specific language within a general-purpose language we can target both programmers and non-programmers as our audience. The embedded scripting language can be tailored to be completely domain-specific and easily understandable by non-programmers while programmers can use both simultaneously in order to write the required AI. Since the scripts written with the embedded scripting language are first-class objects of the host language, programmers can use the latter to manipulate them like any other data type. This gives rise to powerful script generators where the host language acts as a meta-language which queries the game state, takes certain clever decisions and then generates the appropriate embedded language scripts. Finally, the embedded approach maintains the separation of scripts from game logic, allows for scripts to be interpreted in real-time and can also permit third-party programmers to plug in their code if the right interface is provided. Hence we believe that the embedded approach provides a suitable alternative which is in par to other scripting methods.

As case studies to evaluate the approach, we have designed and implemented two games (i) a reflex game 4Blocks; and (ii) a web-based strategy game Space Generals. 4Blocks (see Figure 1) is based on the game Tetris. Here, randomly-generated, bricks of varying shapes appear on the upper side of the screen and fall downwards at a constant rate which increases as the game progresses. The player can perform a series of moves upon each of these bricks such that they are positioned on the bottom of the game area, also known as "well". The objective of the game is to create one or more complete lines. When a line is completed it disappears and any uncompleted lines above it shift downwards to replace it. Completing lines awards the player score points. Space Generals is an online multi-player turn-based strategy game,

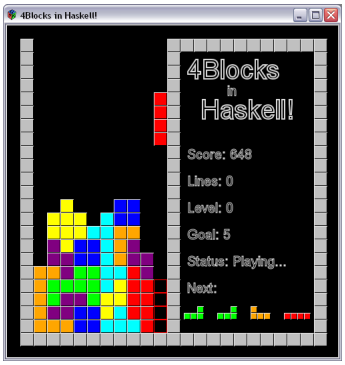

Figure 1: A session of 4Blocks

written in Haskell and the Google Web Toolkit, in which the players, taking the role of generals, try to conquer the universe. Each general commands a number of captains who in turn lead armies. The armies are used to invade other players' domains. Armies cost resources, which can be farmed from planets the general owns. Owning whole planets or even galaxies gives the players additional bonuses.

For both games, domain-specific scripting languages were designed and embedded in Haskell. The languages were extended with general-purpose constructs which allow the scripting language to become expressive enough for programmers to write fixed AI scripts. Furthermore, the host language was used to write adaptive AI scripts which vary according to the game state. Details of the game AI languages and scripts can be found in [3].

5. ISSUES ADDRESSED

Our approach has allowed us to create a scripting method which scores adequately in respect to the desirable attributes mentioned in an earlier section.

Language implementation: Implementing a scripting language using embedded languages is a very easy and straightforward process which should take no more than a few moments. Once the required language has been specified and adequately designed it can be implemented in the host language via the latter's data types and functions without requiring the need to develop a number of tools from scratch as is often done when creating a new language using traditional techniques. Syntax is easily implemented by translating the language's grammar into the equivalent data types. Semantics are also very simple to implement once defined by encoding the rules themselves in an interpretation function. The latter gives meaning to the programs' data type by extracting from them the next action or actions and returning an updated program. For example, implementing the syntax of the domain-specific portion of the scripting language for 4Blocks required just two constructs: one to perform an instruction and another to sequentially compose two or more instructions.

Language updating: Another bonus of this approach is that the language may be evolved at ease both if we need to add any constructs related to the game itself, that is domain-specific such as the ones discussed above, and also if we need to add other idioms, such as those often found in general-purpose languages. Once the constructs have been adequately specified and designed, all we require is to introduce the matching data constructors as syntax and to update the interpretation function accordingly to reflect the new additional semantics. For example, in order to add the possibility of writing AI scripts for 4Blocks we added the following language constructs: *if-then-else*, *while* and *for-this-brick*. Like their general-purpose language variants the first two allow us to query the game state and act accordingly. The third construct allows a program to scope uniquely to one particular brick thus enabling the inclusion of a form of weak exception-handling related to our game.

Language tailoring: Altering the language in order to benefit different user-bases is trivial once the different requirements of these are specified and designed. We can in fact tailor our language towards content-writers such as script writers simply by using Haskell's own module system in order to restrict the constructs of the language exposed. For example, for modules written by script-writers we have exposed only the domain-related constructs *perform* and *sequential-composition*. On the other hand, for actual programmers, we have exposed the full language allowing these to write the required AI scripts.

Language integration: Integrating the scripting language into the game was an easy process since this was achieved by virtue of the embedding process itself, that is by adding the data type representing the language's syntax and by adding the function which gave semantics to structures of this data type. In a way, using our approach, implementing the language is synonymous to integrating it.

Language usability: Our approach has allowed us to strike a balance between accessibility and expressiveness. We selected and implemented domain-specific constructs relating to the game itself thus making the scripting language accessible enough for use by non-programmers in content-scripting. We achieved this by tailoring the language such that it exposed only domain-specific constructs to non-programmers. For AI-programming by programmers we needed to provide a means of querying the game state and running different sub-programs accordingly. We thus added general-purpose programming idioms such as loops and conditionals. Thus our embedded scripting language remained accessible but became also expressive enough to allow us to write AI scripts. Another way by means of which our approach has allowed us to script AI is by making use of the host language itself as a meta-language over the embedded scripting language such that code written in the former examined the game state and generated the required scripts on-the-fly. The Haskell code which achieved this was organised into a number of strategies with different priority. Each strategy queried the game state and if certain requirements were met it was triggered. Once a strategy was selected its corresponding script-generating function was used to generate the relevant script. This function did so based on a number of criteria which depended upon the strategy itself. AI programmers who know the host language may themselves write Haskell modules which generate embedded language code. For 4Blocks, two example strategies which we arbitrarily employed were, to complete four lines whenever the

chance presented itself and to complete lines in order to reduce the well's maximum height as much as possible. We have managed to write AI comparable to a medium-level player but believe that the AI can be improved by adding more strategies or improving the current ones.

Logic/content separation: Since our approach made use of a scripting language and its scripting engine, by nature we separated the scripts from the game engine. Doing this gave us the advantage that the game engine did not need to be recompiled from scratch every time a script needed to be altered. If we needed to update any of the AI written in Haskell or our scripting language, we could change and compile these modules without affecting other game modules directly.

Third-party scripting: The fact that we are using a scripting language enabled us to allow for the possibility of third-party scripting in the form of addons or mods. All we had to do was provide the right API to query the game's state and expose the constructs of the language which we wished to make available to script the game.

General-purpose language available: As we saw when we discussed meta-programming our approach allowed us to make use of the power of a full general-purpose language.

Like general-purpose language scripting and proprietary language scripting our approach is an improvement over game engine integration. Our approach however also improves upon the limitations of the former two. General-purpose scripting languages are intended for programmers only and may not be used by story-writers. Our approach mitigates this by allowing for both a general-purpose language and a game-specific scripting language. In comparison to proprietary language scripting we always have a general-purpose available for programmers to use and our scripting language can be tailored to meet the demands of both programmers and non-programmers easily. Also implementing, tailoring and updating a language with extra features is easy using our approach compared to the other methods using scripting languages. In fact using our approach we can mitigate the main problem these methods face, that of being rejected because of the work required to create and maintain their scripting language. Finally, our approach introduces the very useful feature of meta-programming which we have shown to be very useful in game scripting by our two case-studies. Meta-programming is not a feature which is usually found within the other methods unless it is explicitly introduced.

6. CONCLUSIONS

To the best of our knowledge, our work is the first one which attempts to introduce the technique of language embedding to the domain of game-scripting. By means of it, we have proposed an alternative method to more traditional scripting methods which allows for faster and quicker language creation. If a stand-alone language is still required our approach may be used to quickly prototype a language and test it out before actually going through the implementation process itself. Our approach is thus very useful within the game-development life-cycle.

Our next aim was to see whether a scripting language created using the embedded approach is fit for game-scripting. Using our case-study we have shown how fixed AI scripting makes it possible to use an embedded scripting language to script a game. Moreover, we have improved upon this approach by showing how the host language may be used in conjunction to the embedded language in order to create very powerful adaptive scripts which are generated based upon the game state. Having the power of a general-purpose language as a meta-language over the embedded language is the pivotal aspect of our approach which renders it powerful enough to encode complex AI strategies with ease.

Finally, we have evaluated our method qualitatively and have shown, by means of a number of desirable criteria, that it compares well enough to other scripting methods. The embedded language approach attempts to combine all the advantages more common methods have to offer and provides an adequate alternative method to aid game creators in their endeavours.

7. REFERENCES

[1] Unrealscript language reference, August 2010. http://udn.epicgames.com/Three/UnrealScriptReference.html.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *In International Conference on Functional Programming*, pages 174–184. ACM Press, 1998.

[3] A. Calleja. Embedded scripting languages for game artificial intelligence. Master's thesis, University of Malta, August 2010.

[4] M. Cutumisu, C. Onuczko, M. McNaughton, T. Roy, J. Schaeffer, A. Schumacher, J. Siegel, D. Szafron, K. Waugh, M. Carbonaro, H. Duff, and S. Gillis. Scriptease: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 67(1):32–58, 2007.

[5] C. Elliott. Functional images. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, Mar. 2003.

[6] T. Gutschmidt. *Game Programming With Python, Lua, and Ruby (The Premier Press Game Development Series)*. Premier Press, 2003.

[7] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, June 1996.

[8] P. Hudak. Modular domain specific languages and tools. In *in Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[9] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[10] L. Micallef and G. Pace. An embedded domain specific language to model, transform and quality assure business processes in business-driven development. In *Proceedings of the University of Malta Workshop in ICT (WICT'08)*, 2008.

[11] G. L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

[12] A. Varanese. *Game Scripting Mastery (The Premier Press Game Development Series)*. Course Technology Press, Boston, MA, United States, 2002.