# A NPC BEHAVIOUR DEFINITION SYSTEM FOR USE BY PROGRAMMERS AND DESIGNERS

Eike Falk Anderson
The National Centre For Computer Animation
Bournemouth University, Talbot Campus
Fern Barrow, Poole, Dorset BH12 5BB, UK
E-mail: eanderson@bournemouth.ac.uk

## KEYWORDS
game-bots, behaviour definition, scripting language, virtual machine, mini-language.

## ABSTRACT

In this paper we describe ZBL/0, a scripting system for defining NPC (Non Player Character) behaviour in FPS (First Person Shooter) games. ZBL/0 has been used to illustrate the use of scripting systems in computer games in general and the scripting of NPC behaviour in particular in the context of a book on game development [Zerbst et al 2003]. Many novice game designers have clear ideas about how the computer game they imagine should work but have little knowledge – if any – about how their ideas can be implemented. This is why books on game creation (design, programming etc.), as well as all-in-one game creation systems – especially designed for ease of use and intended for an amateur audience – enjoy great popularity. A large proportion of these books however merely present solutions in the form of descriptions and explanations of specific implementations with inadequate explanations of principles. While this may benefit rapid application development it often does not lead to a deeper understanding of the underlying concepts. The understanding of rule-based behaviour definition through simple scripting in computer games and the development of such scripts by programmers and designers is what we aim to address with the ZBL/0 system.

## INTRODUCTION

Until very recently the major part of the artificially intelligent behaviour displayed by game characters in computer games was hard-coded into the game program itself. Any changes requested by the game's designers needed to be communicated to the game programmers who would spend a large amount of development time implementing these small changes to the game. A much more efficient approach which is now used more and more frequently is to empower designers to implement those changes themselves by making games more extensible and easily modifiable and by providing designers with the tools to extend and modify the games. As a side effect, developers have realized that this also enables players to modify a game themselves which adds value to a game and dramatically adds to its shelf-life (see Table 1). The question that now arises is how this extensibility can be achieved. This is especially important when it comes to the modification of the NPC (Non Player Character) behaviour in those extensible games. In some cases where no hard-coded solutions are used the NPC behaviour is generated by project-specific proprietary software tools, other games use commercially available middleware systems and some games use a scripting language of some sort. Scripting removes a large part of the – previously hard-coded – internal game logic from the game engine and transforms it into a game asset.

| **Are there any truly *good* reasons to build an Extensible AI into your game?** | |
|---|---|
| **Absolutely!** | 40% |
| **Sure!** | 23% |
| **Maybe.** | 29% |
| **No way!** | 4% |
| **Never!** | 2% |
| **Other.** | 1% |

Table 1 – computer game extensibility reasons poll (source: http://www.gameai.com)

This allows the game to be modified without the need for the game code to be recompiled, a task that can be accomplished by a game designer alone. "Parallel development" becomes possible, which means that the programmers' time is freed up as they no longer need to concern themselves with design elements which designers can now manipulate themselves with scripts [Huebner 1997]. However, a scripting language that is supposed to be used by non-programmers as well as by programmers needs to be designed accordingly. It is likely that for some game designers this will be the first programming language that they encounter so it is only logical that it should embrace some of the methods used in introductory programming languages.

## THE RATIONALE BEHIND THE ZBL/0 SCRIPTING SYSTEM

The command syntax of the ZBL/0 language is similar to that of related procedural languages like C [Kernighan and Ritchie 1988], Pascal [Wirth and Jensen 1974] and especially PL/0 [Wirth 1977]. The ZBL/0 language only supports a limited set of control structures (simple iteration, condition/alternative and sequence) and the definition of simple procedures. In that respect, ZBL/0 can be counted in the family of mini-languages (toy languages – see Figure 1) used in teaching [Brusilovsky et al 1997] and in this role it has been used as a reference system to illustrate the development of NPCs [Zerbst et al 2003] for FPS (First Person Shooter) games (see Figure 2). Like other mini-languages ZBL/0 provides a task specific set of instructions and queries which allow users to take control of virtual entities acting within a micro world. In the case of ZBL/0 the scripting system and programming language were designed specifically with the definition of NPC behaviour in FPS games in mind which is reflected in the functions and procedures of the language. The use of computer games as the environment for a mini-language programmed NPC is not a new idea. There are several examples of games – most of which are available on-line

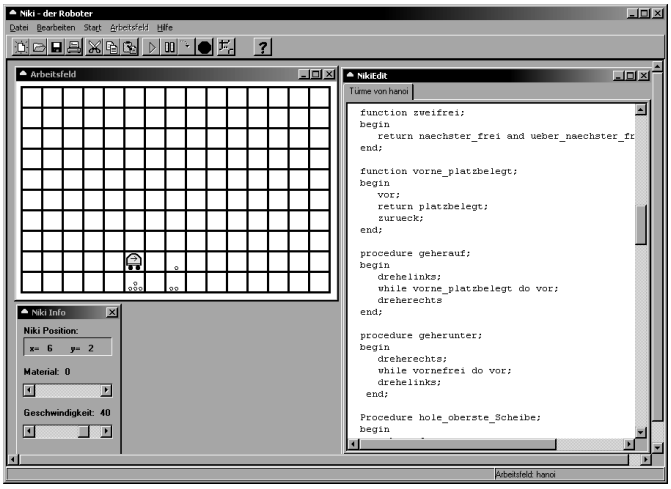


Figure 1 – Mini-language system "Niki the robot" which partially inspired the development of the ZBL/0 language (http://www.hupfeld-software.de/niki.php)

like Robocode [Li 2002], Crobots, Jrobots or GUN-TACTYX [Boselli 2004]. In such games the player interaction is limited to the programming of virtual entities that play the games. In addition to the use of ZBL/0 as an educational tool, the development of the ZBL/0 system is our first step towards the development of a generic behaviour definition system for artificially intelligent entities in computer games. We are using it to explore various system architectures for integrating virtual machines into applications – simple games and more complex game engines – that allow scripts to be executed and interpreted in real-time. We have also used the system as a test-bed for interfacing behaviour-definition systems with computer games.

## SCRIPTING LANGUAGES FOR BEHAVIOUR DEFINITION IN GAMES

Many developers use well established existing generic scripting systems or permutations of these systems (modified according to the game's requirements) to add scripting facilities to their games. A popular choice for building the scripting solutions in games is the scripting language Lua. Lua is a generic programming language which was originally designed to be used to extend programs by adding various scriptable features which is why the creators of Lua have dubbed it an "extensible extension language" [Ierusalemschy et al 1996]. Most of the other mature scripting languages which can be embedded in computer game engines are generic, i.e. not specialised for specific tasks [Varanese 2003]. A different approach which is also frequently used is to have proprietary purpose-built scripting languages that are dedicated to a single game, like the scripting languages QuakeC in Quake, UnrealScript in Unreal or Scrit in "Dungeon Siege". When used to define game character behaviours, in simple cases the sole use of scripts is the initial configuration of the NPC behaviours. These initialization scripts [Tapper 2003] are the simplest form of scripts. During program runtime they are usually only executed once, at program start-up, while the application is initialising, setting internal program parameters to the values in the script. These scripts are often nothing more than lists of values, sometimes using additional syntactic elements to make them easier to read and edit. In more complex event based scripting systems, the occurrence of an event within the game triggers the execution of a script or part of a script. This means that scripts do not run in a pre-defined order but rather when a specific situation in the game-world has occurred. Some of these scripting systems use events that are built into the game engine as predefined events and scripts only define the event handlers and possibly additional conditions that may influence the

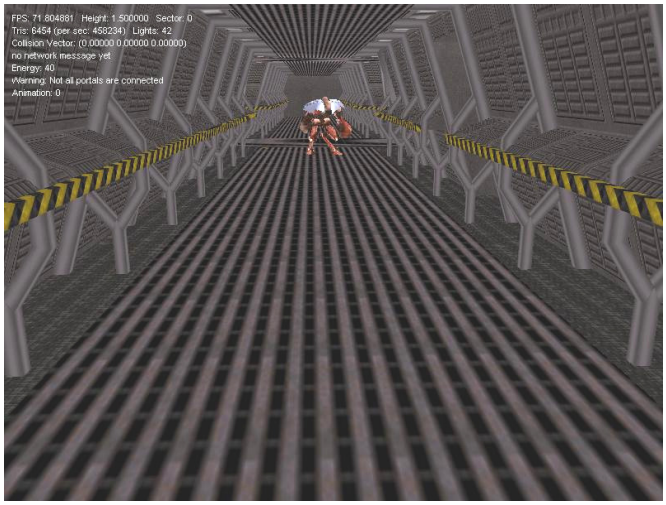


Figure 2 – sample game implementation "Pandora's Legacy"

trigger mechanism. More sophisticated scripting systems first define the triggers and the situations in which they should act on events in addition to the event handlers themselves. These also include rule-based scripting systems which can be used for the definition of domain knowledge in expert systems, an example of which are intelligent NPCs in many computer games. The most complex scripting solutions are programs that use high-level abstract descriptions to define complex behaviours. A scripting system that controls the behaviour of autonomous agents in a virtual game world usually exists on two levels. The higher level is a scripting language that is often modelled on "traditional" procedural, functional or object oriented programming languages. The lower level is the corresponding scripting engine which interfaces with the game. Some of these systems will execute scripts in a continuous loop, constantly (re-)evaluating the current situation within the game. Other systems will execute a script only once and any kind of repeating operation, to be executed by the scripting system, will have to be implemented as a looping operation within the script itself. An example of the latter is our ZBL/0 scripting system. Scripting engines of this kind can take the form of an interpreter which translates and executes scripts at run-time. Alternatively it could be a virtual machine, executing scripts that have previously been translated into an intermediate code by a compiler. Both forms of scripting system provide the same benefits to games, as both allow the alteration of NPC behaviour by modifying a script program. This means that the game application itself does not have to be recompiled for the changes to the game's NPCs to take effect.

### Design Issues

While this is clearly advantageous for game development, for computer game developers to truly benefit from any kind of scripting system it has to be designed to be intuitive, i.e. the scripting language must be easy to learn and possibly easier to use than traditional programming languages. One way this could be achieved would be by making it as similar to a natural language as possible. It is our belief that a close resemblance of a behaviour definition programming language to natural language as suggested by Funge [Funge 1998] may easily prove counterproductive. This is because natural languages are context sensitive and contain too many ambiguities which require additional specification to clarify problems and to resolve these ambiguities. We think that the additional effort required to do this would negate all the benefits gained from the use of a natural language structure in the first place. Moreover, linking a programming language's structure intrinsically to a specific natural language would make it much

more difficult for non-native speakers of the natural language to write meaningful computer programs, while it would become practically impossible for programmers who do not know the natural language to write programs at all. Providing multi-language versions of a programming language is unrealistic, as the language would have to be modified according to the structure of each of the supported natural languages. We are also convinced that the notion that a traditional programming language may be too complex for non-programmers to use is incorrect. Robert Huebner's [Huebner 1997] case study of how scripting support was implemented in the game "Jedi Knight: Dark Forces" describes the C based proprietary scripting language COG. The similarity of COG to the programming language C not only simplified the development of the language but it also made it easier to learn and understand for the designers – non-programmers – who used COG for the creation of the game. He concludes that the design was so successful that designers managed to generate scenarios which would have appeared inconceivable and very hard to realize if it had not been for the COG scripting system. Further evidence for this can be found in the film effects industry where many artists have been using complex scripting systems for many years. For many designers the use of a scripting system will be the first time they are exposed to a programming language. An important consideration in the design of a programming language for novice programmers therefore is the analysis of how many of these non-programmers will go about using this language. Poiker [Poiker 2002] explains how novice programmers write programs employing a mixture of "copy and paste" with "trial and error". For this reason, programming languages that are supposed to be used by novice programmers need to have a WYSIWYG (What You See Is What You Get) character with program source code being able to deliver predictable results. In the context of the novice programmer's introductory programming language, McIver and Conway [McIver and Conway 1996] have identified seven "deadly sins" and design principles and their potential problems and benefits. They argue that a language which has too many different features ("more is more") or too few features ("less is more") or which contains too many syntactical "false friends" ("grammatical traps", "violation of expectation", "excessive cleverness") would make it very hard for users with little programming experience to comprehend the language and to understand what a program does. However, McIver and Conway conclude that the ideas they present can only be taken as a guide – not a general solution – and that ultimately the success of the language design can only be measured through user feedback.

## THE ZBL/0 SYSTEM

The requirements for the ZBL/0 scripting system were straightforward:

- The system was to be used to define NPC behaviour as an extension to computer games of the FPS genre.
- The NPCs defined by the language only needed to support deterministic behaviour.
- No complex datatypes or control structures needed to be implemented as the system was supposed to be used to demonstrate general concepts of NPC behaviour scripting in the context of a book on computer game development [Zerbst et al 2003].

Consequently the development of the system from conception to first use was achieved in a very short period of time. The first fully working prototype for the ZBL/0 system for example was completed over a period of little more than a fortnight. ZBL/0 [Anderson 2003] is a very simple scripting language for the definition of game-bots. The ZBL/0 system consists of a compiler for game-bot programs (NPCs) written in the ZBL/0 language and a robust virtual machine that can be integrated into any game engine.

| | | |
|---|---|---|
| const | do | else |
| function | if | return |
| then | var | while |
| | | |
| alive | armour | back |
| backstep | blocked | crawl |
| danger | die | duck |
| face | find | fire |
| front | health | idle |
| initialize | jump | jump_back |
| jump_left | jump_right | |
| jump_up | left | memorize |
| object | object_ahead | |
| obstacle | owns | respawn |
| right | rnd | spawn |
| spawned | step | strafe_left |
| strafe_right | target | |
| target_ahead | target_alive | |
| target_armour | target_health | |
| turn | turn_left | turn_right |
| use | using | |

Table 2 – ZBL/0 keywords (instructions & intrinsic functions)

ZBL/0 is based on the PL/0 model programming language [Wirth 1977] and therefore belongs to the PASCAL family of programming languages. There is only one variable datatype in ZBL/0 which can be used to store numerical values (integer as well as floating point). The function set for controlling bots is intrinsic to the ZBL/0 scripting language, i.e. built into the language (see Table 2). As a result they do not have to be enabled by means of inclusion of a standard library of functions. This intrinsic function set consists of 45 functions representing actions and sensor queries that can be performed by an NPC in FPS games like turning towards an opponent, moving in a specified direction or firing a weapon. The function identifiers are self explanatory for easy understanding. The current version of the language allows functions to be user-defined, but function parameters in user-defined functions are not supported. Instead they have to be emulated through the use of global variables. The ZBL/0 system uses a parallel stack-based virtual machine – the system is multi-tasking and allows more than one ZBL/0 program to run simultaneously. Run-time errors in ZBL/0 programs result in the termination of the game-bot program but do not affect the execution of the virtual machine within its host application. The ZBL/0 virtual machine is self-contained and accessible from the host application solely through a fixed interface, the ZBL-API (Application Programmer Interface). The interface to the ZBL/0 virtual machine provides games with the ability to associate NPC functionality with in-game functions for actions which would be expected to be performed by a player of these games, therefore allowing NPCs to compete with human players on a level playing field. Once a ZBL/0 program has been loaded into the virtual machine only a single function-call to the API is required for each main program loop to execute the game-bot programs. The simplicity of the system lies in the fact that none of the game-bot functions are provided by the language as such. Instead they need to be implemented within the game engine – the host application – and mapped to the corresponding intrinsic function identifier in ZBL/0. The game engine itself does all the work while the script only ties together the different game engine components that provide the NPCs with functionality. A side effect to this is the ability of the system to be adapted to games of different genres (see Figure 3). The function bindings between the host application and the ZBL/0 virtual machine are realised using the multiple-inheritance functionality of the C++ programming language [Stroustrup 1997]. Objects of a game-bot class can be registered as NPCs with the virtual machine. This game-bot class is created by inheriting player functionality from a player-class in the application and the game-bot interface from an abstract class which is part of the ZBL-API. This abstract class provides a number of methods

```
# a moderately sophisticated CycleBot
var direction;
function random;
var r;
{
  r = rnd 20;
  if r > 5 & r < 15 then
    return 1;
  else
    return 0;
};
{
  spawn;
  direction = random;
  while alive = 1 do
  {
    if blocked front = 0 then
    {
      step;
    };
    else
    {
      if blocked left = 0 then
      {
        if blocked right = 0 then
        {
          if direction = 1 then
          {
            turn_left;
          };
          else
          {
            turn_right;
          };
          direction = random;
        };
        else
        {
          turn_left;
        };
      };
      else
      {
        if blocked right = 0 then
        {
          turn_right;
        };
      };
      step;
    };
  };
}.
```

Table 3 – a simple ZBL/0 game-bot script for a Tron-like "lightcycle" game (see Figure 3)

that are equivalent to the intrinsic functions of the ZBL/0 language. An implementation of these abstract methods in the inherited class then allows ZBL/0 programs in the virtual machine to control a game-bot character in the application. However therein also lies the main weakness of the ZBL/0 system, as any NPC script – no matter how well designed – cannot perform well if the NPC related functions of the game engine do not work well. Also, while this method makes it very easy for the virtual machine to execute functions within the host application it also limits the extensibility of the ZBL/0 system, as the type and number of the functions that can be registered with the virtual machine is fixed by the ZBL-API. On the other hand, this system allows the designer to create effective NPCs through the combination of a small number of simple functions (see Table 3).

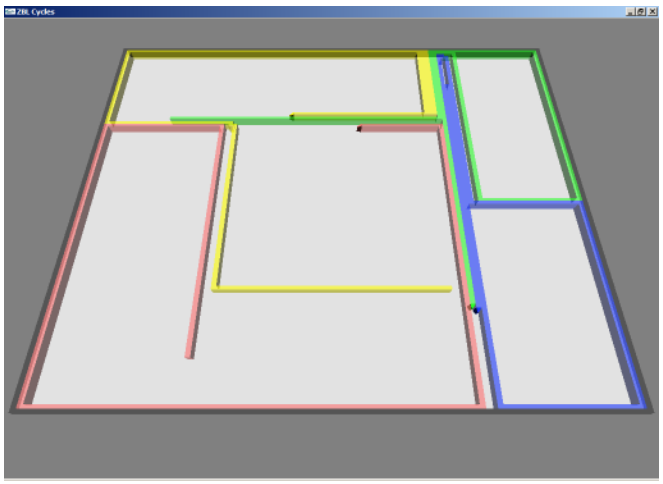

Figure 3 – four ZBL/0 "lightcycles" competing for survival in the demonstration application

## A ZBL/0 Example Program

As an example for the capabilities of the ZBL/0 system as well as a sample to demonstrate the integration of the ZBL/0 virtual machine into a computer game we have created a version of the "lightcycle" racing game (see Figure 3) that featured in the 1982 film Tron (http://www.imdb.com/title/tt0084827). Players in the game move their "lightcycles" across the playing field, dragging growing walls of light energy behind them. The aim of the game is for players to survive as long as possible by avoiding collision with walls while at the same time trying to force other players to collide with walls by reducing their freedom to manoeuvre. In our version of the game all players are controlled by scripts written in the ZBL/0 language (see Table 3). The sensors of the scripted players allow them to test the path immediately in front of them for two steps ahead and one step to each side. The strategy they employ to play the game does not involve any planning but is only a small set of simple rules:

1. If the path in front of the "lightcycle" is not blocked, it moves forward (intrinsic function "step").
2. Otherwise if there is no obstacle to the left and no obstacle to the right, the player chooses a random direction (determined by function random) or if there is an obstacle to the right, the player turns left (intrinsic function "turn_left") and moves forward.
3. In the case of an obstacle to the player's left but no obstacle to the right, the player turns right (intrinsic function "turn_right") and moves forward.

The result of this script is an effective player that is perceived to be far more intelligent by onlookers than it actually is.

## FUTURE WORK

The current version of ZBL/0 only provides the basic tools necessary for using the system with game engines that are programmed using the C++ programming language. As experience has shown that the addition of tools is beneficial for the process of program development, future work on the ZBL/0 system will mainly focus on the expansion of the toolkit. The creation of a graphical user interface – in addition to the command-line tools – to complement the language interface of the system by providing a text-editor incorporating a number of intuitive programming aids found in modern program development tools (syntax highlighting, code completion etc.) will be the first goal. Further goals will be optimizations of the compiler, the addition of support for run-time debugging as well as source-level debugging of ZBL/0 programs and possibly the provision of language bindings for other

programming languages than C++. We have used ZBL/0 to test possible architectures and interfaces for the virtual machine of a more generic behaviour definition system for artificially intelligent entities in computer games. This system that we propose is a modular and easily extendable system that will provide game developers with an intuitive method for the creation of game character AI, as well as the tools for doing so. A newer, experimental version of the ZBL/0 system which is still undergoing testing can be dynamically extended through a plug-in architecture which allows external libraries to be integrated with the system's virtual machine. This is an important feature which will also be implemented in our more generic behaviour definition system.

## CONCLUSION

We have presented ZBL/0, a simple behaviour definition system for game characters in FPS games, designed to be used by game programmers as well as by game designers. ZBL/0 is much smaller, more restrictive and far less extensible than many other scripting systems – the language is dedicated to only one genre of computer games and the AI entities that populate them. Following the example of mini-languages the ZBL/0 language is based on a traditional programming language which has been reduced to the simplest features to make the system easily accessible for programmers and non-programmers alike. We strongly believe that ZBL/0 is easy to learn and master. For the past fifteen years, artists at the National Centre for Computer Animation have learnt to use scripting languages and have successfully used that knowledge for scripting procedural animation. This has convinced us beyond doubt that the use of scripting systems can be picked up by users with no prior knowledge of computer programming. The functionality of ZBL/0 is entirely dependent on the implementation of the host application, yet it shows how relatively simple methods can be used effectively for NPC creation in computer games (see Table 3). As an additional benefit this also allows the system to transcend its limitations by allowing it to be adapted to other game genres than only FPS games (see Figure 3). Some parts of the ZBL/0 system have shown weaknesses in the original design concept which we intend to address with our future work. For instance the lack of extensibility provided by the method in which function bindings are implemented in ZBL/0 has convinced us that a different approach will have to be used for our more generic behaviour definition system. For similar reasons we believe that the use of mainly intrinsic functions results in the main cause of inflexibility of the ZBL/0 system. An implementation using external libraries to provide the core language with functionality would have made the system much more extensible and flexible. The plug-in architecture that was implemented in the latest version of the ZBL/0 system will be able to deliver a partial solution to this problem. This feature of the ZBL/0 virtual machine will be used in a similar fashion in the creation of our more generic behaviour definition system.

## ACKNOWLEDGEMENTS

I would like to thank the members of the ZFX team (at www.zfx.info) – especially Milo Spirig, Sebastian Pech and Oliver Düvel – for their valuable feedback and for program testing. Their suggestions and comments have been encouraging and very useful for the design of the ZBL/0 system. I also need to thank Stefan Zerbst for "mentioning" that a scripting extension to the game "Pandora's Legacy" might be beneficial, prompting me to design ZBL/0 in the first place. Furthermore I need to extend my gratitude to everyone whose comments and suggestions contributed to this paper, especially Prof. Peter Comninos for inspiration, support and help in the preparation of this paper and Steffen Engel for encouragement.

## REFERENCES

Anderson, E.F. (2003). "ZBL/0 - the ZFX Bot Language". ZFX - 3D Entertainment – http://zbl0.zfx.info

Boselli, L. (2004). "GUN-TACTYX - Historical Background" – http://gameprog.it/hosted/guntactyx/info.php#intro0

Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., and Miller, P. (1997). "Mini-languages: A Way to Learn Programming Principles". Education and Information Technologies 2 (1), pp. 65-83.

Funge, J.D. (1998). "Making Them Behave: Cognitive Models for Computer Animation". PhD Thesis, University of Toronto

Huebner, R. (1997). "Adding Languages to Game Engines". Game Developer, Vol. 4(1997): nr 9

Ierusalemschy, R., de Figueiredo, L. H. and Celes, W. (1996). "Lua–an Extensible Extension Language". Software : Practice & Experience, Vol. 26(1996): nr 6, pages 635-652

Kernighan, B.W. and Ritchie, D.M. (1988). "The C Programming Language", Prentice-Hall

Li, S. (2002). "Rock 'em, sock 'em Robocode!" IBM developerWorks: Java technology – http://www-106.ibm.com/developerworks/library/j-robocode/

McIver, L. and Conway, D. (1996). "Seven Deadly Sins of Introductory Programming Language Design". Proceedings of Software Engineering: Education and Practice (SE:E&P'96), pages 309-316

Poiker, F. (2002). "Creating Scripting Languages for Nonprogrammers". AI Game Programming Wisdom, Charles River Media, pages 520-529

Stroustrup, B. (1997). "The C++ Programming Language", 3rd Edition. Addison Wesley

Tapper, P. (2003). "Personality Parameters: Flexibly and Extensibly Providing a Variety of AI Opponents' Behaviors". Gamasutra – http://www.gamasutra.com

Varanese, A. (2003). "Game Scripting Mastery". Premier Press

Wirth, N. and Jensen, K. (1974). "PASCAL - User Manual and Report", Springer-Verlag

Wirth, N. (1977). "Compilerbau", Teubner

Zerbst, S., Düvel, O. and Anderson, E. (2003). "3D-Spieleprogrammierung". Markt + Technik