SCRIPTING BEHAVIOUR – TOWARDS A NEW LANGUAGE FOR MAKING NPCS ACT INTELLIGENTLY

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 language, scripting language, virtual machine.

Abstract

The advent of programmable shaders for real-time graphical applications in recent years has shown that with relatively little effort, great advances in the graphical quality of computer games can be achieved. The introduction of higher level programming languages for the creation of these shaders (*see [Fernando and Kilgard 2003]*) has demonstrated that even better graphical quality for games is attainable. It is our belief that to achieve further improvements in the quality of computer games, a similar approach will have to be taken for the creation of the artificially intelligent characters that populate the virtual worlds found in modern computer games. We therefore propose a universally applicable extension programming language (*AvDL*) for the definition of artificial behaviour for creatures and characters in computer games and possibly in computer animation. The syntax of our language is based on that of the C++ programming language as described by [Stroustrup 1997] and provides for the definition of deterministic, as well as goal directed behaviour for artificial entities. This will make it possible to use our language for different purposes (*the definition of NPCs – Non Player Characters – in games of different genres*) and with a wide range of programs.

1 Introduction

A recent trend in computer games is to make the games extensible by allowing the players, to modify the games according to their needs and preferences. While this can mean a simple

manipulation of the appearance of the virtual game environment, one of the main areas in which games allow modifications of this kind is in the behaviour of the game AI. The method by which the extensibility of most new games is mainly realised is by the use of scripting systems. Scripting can be used to issue commands to the game engine like loading objects, textures and levels, but also for much more complicated tasks like playing animated cut-scenes, directing camera movements or triggering events inside the game world. The use of scripting is also the most common method by which the AI behaviour of a game is extended or modified. Scripting replaces a large part of the – previously hard-coded – internal game logic from the game engine and transforms it into a game asset. Scripts can be used to direct the application of these assets to the game, effectively modifying the behaviour of the game engine and the game itself without the need for the game source-code to be recompiled. With scripts themselves being game content, this means that the game engine only provides a shell, a protected "sandbox" environment within the game engine, for the scripts to operate in. The scripts are used to create the game and its environment without being able to adversely affect the game engine itself.

At this point we should note that we use the terms scripting language and scripting system to describe a system using a programming language which is not compiled into native machine code for execution but rather has its code compiled into an intermediate form for execution within a virtual machine. Others like [Sweetser and Wiles 2005] consider scripting a method for prescribing specific events and behaviour, very much like a film script which cannot be altered. We however refer to scripting when we describe a system which allows the modification of the game logic without the need to recompile the game (*engine*) source code.

Scripting has been used in game development for quite a long time. Access to those scripts however has usually been limited to the game developers, and only in recent years the power to modify games has been opened up to the game players. This new trend has also had a significant impact on the structure of the scripting systems used by game developers. Whereas originally the scripting systems were only used in-house by programmers and designers who had direct access to the programmers in case of any difficulties with the system arose, now they had to be developed to a point where they could be 'let loose' on the general public where mainly non-programmers would use them to modify the game.

1.1 Scripting Systems in Games

Many developers use well established existing generic scripting systems or permutations of these systems to add scripting facilities to their games. Other games have proprietary purpose-built scripting languages that are dedicated to a single game. The various types of scripting systems commonly found in modern computer games are:

initialisation scripts

[Tapper 2003] describes this most simple form of scripting system. During program runtime scripts are usually only executed once, at program start-up, while the application is initialising. In most cases this kind of script is used only to set internal program parameters to the values in the script which is why initialisation scripts are often nothing more but lists of values, sometimes using additional syntactic elements to make scripts easier to read and edit.

trigger-only induced scripts

In 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 trigger mechanism. More sophisticated scripting systems that follow the concept of "Action Languages" as described by [Gelfond and Lifschitz 1998] first define the triggers and the situations in which they should act on events in addition to the event handlers themselves. These trigger-definitions will usually be executed during the initialisation of the scripting system so that these events can be generated by the game engine if all necessary preconditions are met. The conditions for triggering events will then be continuously checked against the in-game situation and if they evaluate as true they will induce the execution of the event handler. Alternatively events can be triggered from within other events. This category of scripting systems also includes 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.

scripts which run like a traditional computer program

Then there are the scripting systems that are modelled on "traditional" procedural, functional or object oriented programming languages. Some of these scripts will continuously loop and (*re-*)evaluate the current situation within the game and will restart execution from the beginning of the script, once the end of the script has been reached. [Anderson 2002] used this type of script to evaluate the use of genetic programming for computer generated game players. Other scripts of this type will execute once only 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 for this is the mini-language like behaviour definition system described by [Anderson 2004].

1.2 programs that modify NPC behaviour

A scripting system of some sort (*established or proprietary*) might seem an ideal solution for generating NPC behaviour, but there is not one single method by which the behaviour of artificially intelligent characters is created. Therefore a solution found for one game is not easily transferable to other computer game productions, which is especially true when it comes to the scripting of NPC behaviour. Some of the different kinds of scripting systems which are used in conjunction with AI in games are quite generic and are not exclusively used for scripting the AI, but also for other tasks within the game. There are some dedicated AI scripting systems that have been used in a number of games and animation systems. In most cases they have been highly specialised for specific genres of computer games or kinds of behaviour that is generated by the system.

The design of a programming language for the definition of artificial behaviour as an extension to a specific game or genre of computer games (*for example 1st Person shooters*) is relatively simple if only deterministic behaviour is involved. For instance, the first prototype for the ZBL/0 system – an educational tool for creating game-bots – was developed over a period of little more than a fortnight (*from conception to first use*). In effect such a language does little more than provide a function binding interface to a game engine for the creation of rule based systems. 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. Unfortunately the specialization for a single genre greatly restricts the reusability of such systems and they are usually proprietary to a specific product or range of products.

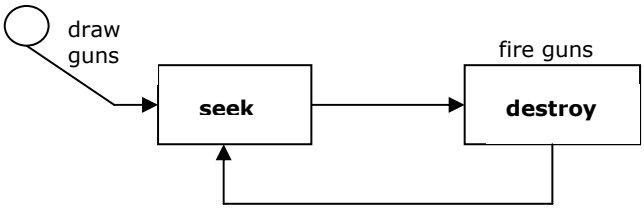
2 Introducing AvDL

A generic system for the definition of NPC behaviour would result in a reduction of the workload for games programmers and therefore simplify game development. When using a behaviour definition system like this, the process of developing a computer game could be further improved through close integration of this system with the tools and programs that are already used for creating computer games, like level editors or 3d content creation applications. In addition to making it easier for a game production team to meet its time and budgetary constraints, such a system would make game development more cost effective by allowing parallel development. Exactly this is what the system we propose aims to achieve through the use of a dedicated software toolkit for the definition of artificial behaviour for creatures and characters with possible applications in computer games and computer animation. At the core of our system resides the Avatar Definition Language (*AvDL*), a dedicated programming language with a compiler suite. AvDL is more than just a game AI scripting language but rather a form of behaviour definition language for virtual entities which is easy to use and offers powerful game AI functions while maintaining the flexibility of traditional programming languages. AvDL programs are executed in a virtual machine which interfaces with a game engine and provides for the definition of deterministic, as well as goal directed behaviours. The language itself only provides the means for defining program flow and some NPC-AI related datatypes and operators. AI functions for use with AvDL can be accessed from a standard library. This AvDL standard library provides its functions in native code (*in the operating system of the host application*) to guarantee the fastest possible execution time. [Huebner 1997] suggests that similarity to an existing programming language can benefit a scripting language. The syntax of the AvDL programming language inherits much from the C family of programming languages and is largely based on the C++ programming language introduced by [Stroustrup 1997]. However, some fundamental elements of the core language have changed in relation to C/C++:

- there is only one numeric datatype – the scalar datatype which can take floating point values as well as integer values
- there are additional AI related datatypes and operators
- pointers are not used (*no direct memory access mechanism exists*)
- function parameters are passed by reference unless specified otherwise
- there is no multiple inheritance of classes
- there is no function inlining

- there are additional control structures

```
action drawGuns, fireGuns; // actions – functions in host program
state player
{
  onentry();    // define onentry "constructor" method
  void seek(), destroy; // declare seek state, followed by destroy
  destroy: fireGuns, seek; // destroy uses "fireGuns"
};
player::onentry() // onentry method
{
  drawGuns; // initial action when state machine is entered
}
void player::seek()
{
  ...  // do something (search for target)
}
... // a function (main program) that uses a "player" state machine
{
   player myPlayer;
   ...
   myPlayer = setstate seek;  // initialize state machine
   ...
}
```



When the state machine is initialized, the player draws his guns (*onentry*) and enters the "seek" state. When the player finds his target, a state transition from "seek" to "destroy" takes place and the player will try to destroy his target (*by firing his guns*). After the target is destroyed, the player's state switches back to "seek" for other targets.

Table 1: A finite state machine for a simple FPS game-like scenario

Despite the danger of leading to confusion, AvDL retains most of the more abstract elements of C/C++. Some macro-like syntactic synonyms (*aliases*), to be substituted by a pre-processor, are integrated into the language definition as alternative means for expressing programs. This should present non-programmers with a simpler and easier to read alternative without the language having to sacrifice its similarity to its C-like syntax. As [Brockington and Darrah 2002] point out, every scripting system will be used for purposes unforeseen by the system's designers and users will tend to bend the system close to its breaking point. A scripting system therefore has to be as flexible and extensible as possible while at the same time being robust and maintaining run-time stability to avoid the kind of catastrophic failure which could disrupt the game engine beyond the point of recoverability. To achieve this stability all variable datatypes in AvDL are auto initialising. This means that a variable that is declared will initially be provided with a default value unless otherwise specified in the AvDL program. This simplifies the declaration process and helps to avoid errors within an AvDL program which may be hard to debug.

One major difference between AvDL and languages like C and C++ are the game AI specific datatypes in AvDL. AvDL provides a datatype for the definition of states (*finite as well as fuzzy*) which allows for the definition of hierarchical state machines (*see [Fu and Houlette 2004]*). State machines are a tried and tested AI technology for the definition of deterministic behaviour, which has proved suitable for many kinds of computer games. The state datatype allows for simple state machines to be defined using basic AvDL instructions without any libraries (*see Table 1*). AvDL state machines are initialised by using the unary "setstate" operator to set the initial state. Once this has been set, the state machine will start its execution. Goal-directed behaviour is one of the simplest forms of non-deterministic behaviour. A goal is the end-state of a set of goal-directed actions. [Orkin 2004] describes how goal-oriented action planning can be used to generate the path that the system needs to take to reach this end-state. Goal-directed behaviour can be achieved in AvDL by using the "goal" datatype (*see Table 2*). Goal-oriented action planning requires the use of a number of specialized AvDL operators:

- the operator for the planning of action sequences is the "plan" operator which operates on goals
- the unary operator "reached" is used for testing a goal for completion

```
action actionX,actionY; // actions – functions in host application
goal goalX  // goal with two sibling sub-goals (brother and sister)
{
    brother: actionX==FALSE; // reached when actionX==FALSE
    sister,10:actionY==TRUE; // reached when actionY==TRUE
};
…
{
    goalX mygoal;         // declare a goal of type goalX
    mygoal = plan goalX;  // develop an action plan
    …
    while(reached(goalX)!=TRUE) // wait for goal to be reached
    {    // if plan becomes invalid then replan
         if(reached(goalX)==NULL) mygoal = plan goalX;
    }
    …
}
```

Table 2: Goal directed behaviour definition in AvDL

AvDL programs can be enabled to directly call functions that are defined within the host application. These function bindings are created by using the AvDL datatype "action" which maps AvDL actions to functions in the host application. In addition to functions, data in the host application can be bound to variables in AvDL programs by mapping variables in the host application to variables in the AvDL program through the AvDL API.

3 Work In Progress

The work presented thus far is very much work in progress – we are currently working on the implementation of a prototype system. It is important to note that the current language specification is not final and may still be subject to change. It is also quite possible that in reaction to feedback from users of the AvDL system, changes to the AvDL programming language might have to be made if so required by the users. We endeavour to complete the implementation of the core functionality of AvDL – compiler and virtual machine – in the current stage of the project. In order to test a number of implementation ideas for our AvDL behaviour definition language we have designed and implemented the ZBL/0 scripting system (*see [Zerbst et al 2003]*). ZBL/0 is much smaller, more restrictive and far less extensible than

AvDL. 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. We used ZBL/0 to test possible architectures and interfaces for the AvDL virtual machine. 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. Based on our experiences with this system we believe that a similar approach should be taken with the AvDL virtual machine. To be able to run as a plug-in the AvDL system will have to be self-contained and accessible through a fixed interface. This is also the case with the ZBL/0 virtual machine which is located entirely within a static library and is only accessible through the ZBL-API. The ZBL/0 system can by dynamically extended through a plug-in architecture which allows external libraries to be integrated with the system's virtual machine. This important feature of the ZBL/0 virtual machine will be used in a similar fashion for the creation of the AvDL standard library. This AvDL standard library will provide standard functions and define appropriate compound datatypes for solving a variety of game AI tasks. The functions contained in the standard library will enable a user to define an avatar's domain knowledge, i.e. the avatar's perception and understanding that is necessary for "surviving" in the virtual world it occupies. These functions will adhere to the standards suggested by the IGDA AI Interface Standards Committee (*see [Nareyek et al 2004]*).

4 Conclusion

Our proposed system provides a synthesis of the functionality of a wide range of different technologies that are used in the development of NPCs in computer games. Different concepts – deterministic behaviour as well as goal-oriented behaviour – are combined within a single application which is generic, i.e. not limited to a single type or genre of computer game. Although we are only in the early stages of implementing our ideas and therefore have not yet had the opportunity to validate our approach through testing, we are confident that our approach will prove successful. It is our firm belief that our system would be sufficient for defining and controlling all variations of artificial characters that can be found in current computer games.

5 Future Work

The next step in this project is the completion of the AvDL system prototype which will allow us to evaluate the capabilities of the AvDL system. A first step towards this goal is the development of SEAL, a fully working subset of the AvDL behaviour definition language (*see [Anderson 2005]*). Furthermore we will define a standard library of functions for AvDL which will provide functionality for solving the most common problems that occur during the definition of artificial character behaviour. Finally we aim to integrate the AvDL system prototype into existing game engines and applications to prove the suitability of the AvDL system for the game development process.

Acknowledgements

This paper presents a brief summary of the last three years of my work, so first and foremost I would like to thank my supervisor, Prof. Peter Comninos for his help, encouragement and guidance. Additional thanks go to everyone who supported my research, among others Anargyros Sarafopoulos, but first and foremost my family, especially my late grandmother, without whom I would not be doing what I am doing now.

References

[Anderson 2002] Anderson, E.F. (2002). Off-Line Evolution of Behaviour for Autonomous Agents in Real-Time Computer Games. Proceedings of Parallel Problem Solving from Nature – PPSN VII, LNCS Vol. 2439, pages 689-699

[Anderson 2004] Anderson, E.F. (2004). A NPC Behaviour Definition System for Use by Programmers and Designers. Proceedings of CGAIDE'2004, pages 203-207

[Anderson 2005] Anderson, E.F. (2005). SEAL – a Simple Entity Annotation Language. To appear in Proceedings of zfxCON05 2nd Conference on Game Development

[Brockington and Darrah 2002] Brockington, M. and Darrah, M. (2002). How Not to Implement a Basic Scripting Language. AI Game Programming Wisdom, Charles River Media, pages 548-544

[Fernando and Kilgard 2003] Fernando, R. And Kilgard, M.J. (2003). The Cg Tutorial. Addison-Wesley

[Fu and Houlette 2004] Fu, D. and Houlette, R. (2004). The Ultimate Guide to FSMs in Games. AI Game Programming Wisdom 2, Charles River Media, pages 283-302

[Gelfond and Lifschitz 1998] Gelfond, M. and Lifschitz, V. (1998). Action Languages. Linköping Electronic Articles in Computer and Information Science, Vol. 3(1998): nr 16

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

[Nareyek et al 2004] Nareyek, A., Karlsson, B.F.F., Wilson, I., Chady, M., Mesdaghi, S., Axelrod, R., Porcino, N., Combs, N. El Rhalibi, A., Wetzel, B. and Orkin, J. (2004). The 2004 Report of the IGDA's Artificial Intelligence Interface Standards Committee. IGDA AIISC – http://www.igda.org/ai/

[Orkin 2004] Orkin, J. (2004). Applying Goal-Oriented Action Planning to Games. AI Game Programming Wisdom 2, Charles River Media, pages 217-228

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

[Sweetser and Wiles 2005] Sweetser, P. and Wiles, J. Scripting Versus Emergence: Issues for Game Developers and Players in Game Environment Design. International Journal of Intelligent Games and Simulations 4 (1), pages 1-9

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

[Zerbst et al 2003] Zerbst, S., Düvel, O. and Anderson, E. (2003). 3D-Spieleprogrammierung. Markt + Technik