Published in Proceedings of VS-GAMES 2011: 3rd International Conference on Games and Virtual Worlds for Serious Applications, pages 47-54; definitive version available from http://dx.doi.org/10.1109/VS-GAMES.2011.13

A Classification of Scripting Systems for Entertainment and Serious Computer Games

Eike Falk Anderson
Interactive Worlds ARG
Coventry University
Coventry, UK
eikea@siggraph.org

***Abstract*—The technology base for modern computer games is usually provided by a game engine. Many game engines have built-in dedicated scripting languages that allow the development of complete games that are built using those engines, as well as extensive modification of existing games through scripting alone. While some of these game engines implement proprietary languages, others use existing scripting systems that have been modified according to the game engine's requirements. Scripting languages generally provide a very high level of abstraction method for syntactically controlling the behaviour of their host applications and different types of scripting system allow different types of modification of their underlying host application. In this paper we propose a simple classification for scripting systems used in computer games for entertainment and serious purposes.**

***Index Terms*—Data-Driven Game Development, Scripting Language Classification.**

I. Introduction

The computer games industry is still a young industry which continues to have a lot of potential for growth. This is especially so, since games have broken out of the domain of pure entertainment and are now used in a wide variety of different situations. Edutainment or game based learning [1] provide only one flavour of the application areas for computer games technology, or as Zyda notes, "applying games and simulations technology to nonentertainment domains results in serious games" [2]. However, independent of whether a computer game is developed for entertainment purposes only, or as a serious game, it will be created using similar processes and usually from the same technology base (Fig. 1) [3].

One of these technologies that many developers use to create games are well established generic scripting systems or permutations of these existing systems (modified according to the particular requirements of the games that are being created) that add scripting facilities to their game engines and games. Alternatively there are proprietary purpose-built scripting languages that are dedicated to a single game or game engine.

It is a fact that "... language is what gives humans enormous leverage over the universe" [4]. Analogous to this, scripting languages in computer games, which provide control over the behaviour of the application, give the game developer "enormous leverage" over a game's virtual reality.

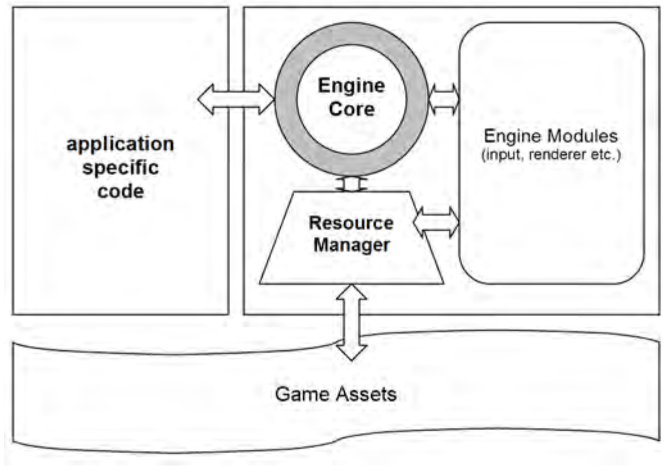
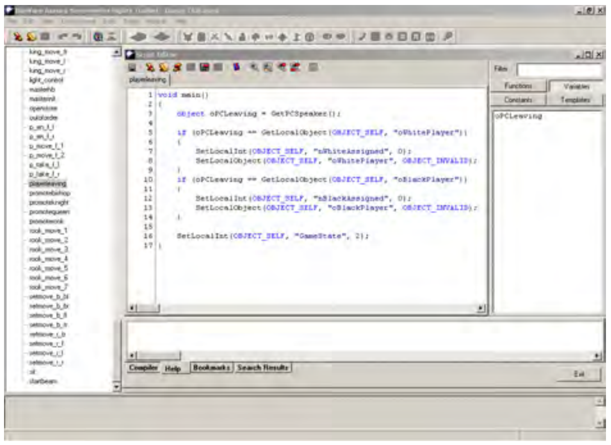


Fig. 1. A typical game engine. If the engine core includes a scripting system, then application specific scripts are loaded as external game assets and there may be no other application specific code.

Consequently it is no surprise that scripting systems are considered one of the most important developer tools that are included in modern game engines [5]. In game development, scripting languages are used within the games themselves (by embedding them within the game engines) or in the tools used for game development – usually in situations where the use of an implementation language such as C++ would be inappropriate [6]. Although scripting has been used in game development for quite a long time [7], access to those scripts has usually been limited to the game developers, and only in recent years the power to modify games has been opened up to the end users, i.e. the game players. Whereas originally the scripting systems were only used in-house by a game's programmers and designers who had direct access to the programmers in case any difficulties with the system arose, now they tend to be developed to a point where they could potentially be 'let loose' on the general public where mainly non-programmers use them to modify the games that they are embedded in.

A. *Game Modification and Serious Game Development*

Computer games that can be modified by their user community enjoy great popularity. As a result some of the



```
void main()
{
    object oPCLeaving = GetPCSpeaker();

    if (oPCLeaving == GetLocalObject(OBJECT_SELF, "oWhitePlayer"))
    {
        SetLocalInt(OBJECT_SELF, "nWhiteAssigned", 0);
        SetLocalObject(OBJECT_SELF, "oWhitePlayer", OBJECT_INVALID);
    }
    if (oPCLeaving == GetLocalObject(OBJECT_SELF, "oBlackPlayer"))
    {
        SetLocalInt(OBJECT_SELF, "nBlackAssigned", 0);
        SetLocalObject(OBJECT_SELF, "oBlackPlayer", OBJECT_INVALID);
    }

    SetLocalInt(OBJECT_SELF, "GameState", 2);
}
```

Fig. 2. BioWare's Aurora Toolkit is the game editing toolkit for the game 'Neverwinter Nights'. It not only provides methods for building the game environment and placing objects and virtual actors for game extensions but also the means for defining the actions of these actors and the conversations that a player can have with them using various scripting tools which are embedded within the toolkit.

most advanced script development aids for extending existing games can be found in exactly those computer games. These tools, often coupled with world editors in the form of an integrated development environment (Fig. 2), are now included in many game releases, effectively making them additional game content [8]. This is of special interest for developers of serious games who generally do not have the financial resources that entertainment game developers have [9], as it allows them to use the same technology employed in the creation entertainment games for a small fraction of the price. An example for this use of entertainment games as the infrastructure for serious games is 'The History Game Canada' (http://historycanadagame.com), which is a modification of the entertainment strategy game 'Sid Meier's Civilization III' [3].

One of the most mature and most widely used game engines that allow complete game modification through scripting is the Unreal game engine. Not only has this engine been used in a large number of commercially successful games, but the extensibility of these games has resulted in the production of many additional modifications to these games by end users, using UnrealScript, the scripting language used by the Unreal engine [10]. Serious games created using the Unreal engine are the 'Virtual Egyptian Temple' (Fig. 3) [11], [12], and derived from this the game 'Gates of Horus' [13], that let players explore a hypothetical Egyptian New Kingdom period temple.

In this paper we discuss the use of scripting systems in computer games, in an attempt to establish a useful classification of this technology. **Section II** discusses data-driven architectures for computer games, focussing on the manifestation of the data-driven design philosophy in the use of scripting languages. **Section III** provides an overview of scripting systems and scripting languages with a specific focus on existing solutions using generic embeddable scripting

Fig. 3. The Virtual Egyptian Temple modification of the game Unreal Tournament 2004 [11].

languages for use in computer games and **Section IV** presents our proposed classification of scripting for entertainment and serious computer games. Finally, in **Section V** we present our conclusions and discuss potential avenues for future work.

II. Data-Driven Computer Game Development

In software development, in general, the use of a data-driven architecture usually means the distinction of an application's core components from application specific code. The former are code elements that may be reused unchanged in other applications, whereas the latter indicates code or data that is unique to the individual application. This implies an abstraction of the application's internal logic from the data which is used to define the application's behaviour [14], separating the definition of the application's make-up from the application's core functionality, which becomes effectively 'policy free'.

Being 'policy free' means that while the application's core provides functionality which entails only the means for the creation of an application, i.e. the building blocks from which a comprehensive application can be constructed, it does not, however provide the application's functionality itself. In simple terms, it provides the 'how to do', but not the 'what to do'.

If the application is a computer game, a data-driven architecture results in games driven by a game engine [10], [15] (Figure 1). This allows developers to make a clear distinction between engine (code) and game code, the former being the core elements that may be shared among several distinct games and the latter being the code that is unique to the specific game. As most of the game specific logic is no longer an intrinsic part of the core source code, in general a data-driven game engine is highly reusable and believed to be cost efficient [16], enjoying a relatively long shelf-life.

There are different layers of abstraction that define the make-up of the data part of data-driven games, but borders between these layers are not strictly defined and vary depending on the individual implementation. In its simplest form, the game specific data can take the form of source code which can be linked with the game engine core. A higher level of abstraction on the other end of the scale is to store this data as an external game asset. Game assets are those elements of a game that are loaded into the game engine at run-time to provide the content of the game, including elements which are created by designers and artists like 3D models, textures and animation or sounds and music.

In game development data-driven design is often understood as a way to empower artists and designers to independently modify game logic without a programmer's help or intervention [17], requiring this to be accomplished without the need to recompile parts of the game program's source code. The methods used to achieve this are the same ones that also allow external game modification.

The Achilles heel of this high level of data-driven design in any computer application is the fact that an outsourcing of product specific data into an external asset can allow malicious users to effectively hijack the system by modifying those external resources or by replacing them with their own resources, providing these users with unfair advantages [18]. This type of misuse of scripting systems, however, can be easily prevented if the application properly verifies the integrity of its external resources before they are used. In the case of computer games, the modification of external assets can even be desirable, which is evident in the many extensible games that allow users to make their own modifications.

A. *Game Extensibility and Modification*

Over the past decade there have been many games that have been created in a way that allows the players to directly modify the games. This "modding" of games [19] goes from the simple extension and addition to existing games up to the creation of completely new games. This has been supported by the games industry through the publication of the same tools used by the game designers for the creation of the games themselves. By exposing the end-user, i.e. the players, to the tools allowing them to extend and modify the games themselves and by assisting them with any game modifications they intend to make, the developers add value to a game and dramatically increase its shelf-life. To simplify this, some games provide extensive software interfaces into the game engine, allowing parts of the games to be reprogrammed by direct manipulation of the game code or through plug-ins, however, the method by which the extensibility of most modern games is realised is by the use of more or less complex scripting systems. This is because a scripting system in which the script has complete control over the behaviour of the application that it is embedded in is the ultimate implementation of a data-driven design.

Varanese [20] explains and discusses in detail how scripting is used in combination with computer games and how scripting systems can be embedded within computer games. Scripting can be used to issue commands to the game engine, such as loading of 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 worlds. It removes a large part of the – previously hard-coded – internal game logic from the game engine and transforms it into a game asset. Scripts themselves 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 "a form of artistic content" [18] for games, this means that the game engine only provides a shell, i.e. a protected 'sandbox' environment for scripts within the game engine. Scripts operate within this 'sandbox' with the scripts creating the game and its environment without being able to adversely affect the running of the host game engine itself.

III. Scripting Languages, Scripting Systems and Computer Games

The Oxford Reference Online defines a scripting language as "a programming language that can be used to write programs to control an application or class of applications, typically interpreted" [21]. This is only one of many different definitions for scripting languages and this very broad definition encompasses a vast range of programming languages which is – unfortunately – not very helpful.

When it comes to games, some consider scripting a method for prescribing specific events and behaviour [22], very much like a film script which cannot be altered. We however refer to the terms scripting language and scripting system when we describe a system using a programming language which allows the modification of program logic without the need to recompile the application (game engine) source code.

Scripting languages are used to provide a control interface for combining different components into a single whole, which is why they are also "referred to as glue languages or system integration languages" [23]. They are "meant to be easy to program in" [24], often at the expense of run-time performance. As such, scripting languages provide an additional layer of abstraction on top of components (or programs) usually written in a high-level programming language. This abstraction, combined with the fact that modern scripting languages such as Python [25] have a lot in common with traditional system programming and implementation languages such as C and C++, makes scripting languages a form of VHL (Very High Level) programming languages [26].

Scripting systems have a wide range of applications and can appear in many different forms, depending on the area of application. Some of the simplest scripting systems are the sophisticated command-line interpreters related to UNIX shells such as Ksh [27], their main task being to tie together external programs into a unified construct. Their scope can be greatly enlarged through the use of file processing languages such as AWK [28], which form the next higher level of scripting system. Different from these standalone systems are integrated scripting systems such as MEL (Maya Embedded Language) [29] that control a single application from the inside, often requiring very little overhead from the application's side for executing scripts, although this is not the case with MEL (see above). Embedded scripting languages are often found in applications for use by non-programmers, i.e. in programming terms “less-skilled personnel” [4] or “semiprogrammers” [30] for whom programming is not an intrinsic part of their job-description. They include DSLs (Domain Specific Languages) [31] that can also take the form of macro-based languages that are embedded within an implementation language to be actually translated into native code and linked with its host application [32], which is a technique considered to be a good use of preprocessor macros [33], [34].

While many scripting languages are interpreted, this is not generally the case. Immediate interpretation of scripts which are directly analysed and executed statement by statement is an expensive operation. To achieve a better performance it makes sense to compile script programs, however, not into a frozen executable in native machine code, but rather into an intermediate form for execution within a virtual machine. Scripts that are not interpreted directly but pre-compiled into intermediate interpreter code, running on a virtual machine, can attain considerable performance improvements over those that are interpreted statement by statement, while also preventing some otherwise hard to detect run-time errors by catching them during script compilation. If that compilation happens to be performed on-the-fly, i.e. if the compiler is integrated into the virtual machine as a kind of script preprocessing step, this process is hidden from the script programmer, providing the illusion that the script is directly interpreted. This is a technique employed by some of the more advanced scripting languages with features that are very close to those of popular implementation languages, showing that they can be a viable alternative to those very same ‘conventional’ programming languages [35].

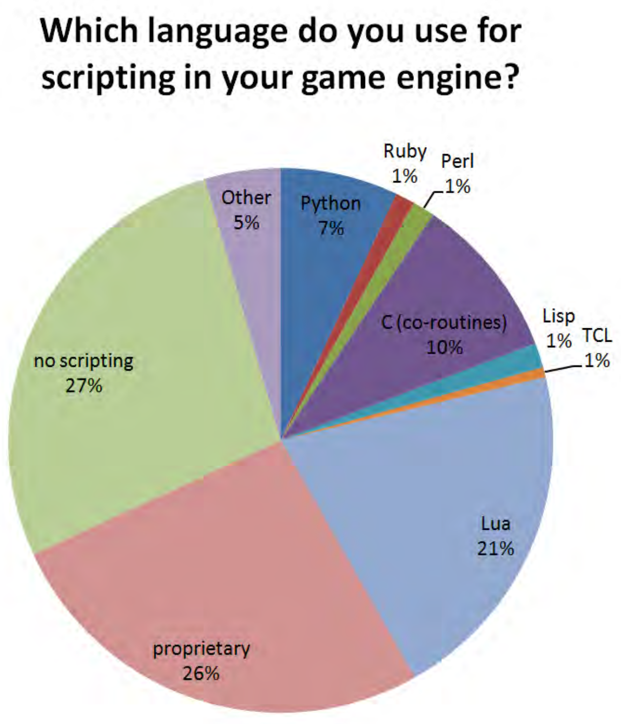


Fig. 4. A 2003 computer game scripting poll conducted at the game development website www.gamedev.net – frequented by many game development professionals, as well as amateur developers – suggests that nearly 75% of game engines in development at the time included some form of support for game modifications through scripting systems, over a third of which were proprietary solutions.

A. Game Design Improvement Through Scripting

Whereas only a few years ago the majority of scripting solutions used in computer games were proprietary languages (Fig. 4), the trend has now shifted towards the use of generic scripting solutions of which some have been designed explicitly for use in computer games. This becomes evident in a recent survey of scripting languages in computer games [36] which focuses on the languages Lua, Python, AngelScript and GameMonkey Script, all of which are embeddable languages that have been used in commercial games. Other than these popular choices for scripting languages in games, there exist a number of less frequently used but mature scripting languages which can be embedded in computer game engines. Most of these languages are generic, i.e. not specialised for specific tasks [20]. Generic languages of this type that are frequently mentioned in the context of game development are the languages Tcl and Ruby. Other languages used with games are the object oriented language Squirrel or the language JavaScript (standardised as ECMAscript – ECMA-262) which has its origins in web-browsers but has since found a wide range of applications, an example for which is the ActionScript language used in the Flash multimedia authoring system, which is also used in game development [37].

This does not, however, mean that the development of proprietary scripting solutions should be avoided at all costs, as Wilcox notes that despite the effort and development overhead involved in the creation of a new scripting system “you are not reinventing the wheel. You are creating a way to concisely express your thoughts in a new language” [4]. Huebner’s case study of how scripting support was implemented in the FPS (First Person Shooter) game ‘Jedi Knight: Dark Forces’ details the development process of a proprietary language for use by the designers of the game [38]. In this study Huebner clearly identifies the benefits of using a scripting system in game development:

- The complexity of the core game engine is reduced as elements of the game logic are taken out of the engine and put into scripts instead.
- The stability of the core game engine is enhanced as a less complex engine design will have fewer vulnerabilities and bugs.
- “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.
- Designers are empowered and given the opportunity to realize more aspects of their designs – this is especially true when the virtual machine can do just-in-time (JIT) compilation of scripts and when the script editor is integrated with the level editor.

While the development of a proprietary game scripting solution places additional burden on the developers of a game engine, it not only allows the scripting language to be domain specific to the type of game that the engine is supposed to provide an infrastructure for, but it also allows the language to provide abstractions for problems that are not usually addressed by generic programming languages, such as syntactic features that specifically deal with game AI. This in turn can simplify the actual development of games.

IV. Comparative Analysis and Classification of Scripting in Games

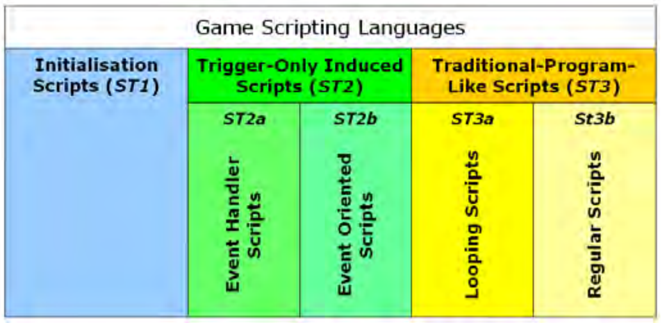


Fig. 5. Classification of scripting systems for computer games.

Just as the term 'scripting' has different interpretations depending on the context in which it is used, there are different types of scripting systems, each working differently and not all of them are suitable for use in computer games. Our classification of the various types of scripting systems is restricted to those found in modern computer games and does not attempt to be complete but rather means to serve as a guide for distinguishing between different script types by their usage. The various types of scrips that are employed in computer games are (Fig. 5):

ST1 – Initialisation Scripts:

$ST1$ initialisation scripts can be considered the simplest form of scripting system [39]. During program runtime scripts of this type are usually only executed once, at program start-up, while the script's host application is initialising. In most cases this type of script is used only to set internal program parameters to the values given in the script which is why these scripts are also referred to as "property scripts" [40].

Initialisation scripts are often nothing more but lists of value declarations that are usually interpreted directly, i.e. scripts that are not require translation into an intermediate format for execution. Sometimes these languages include additional syntactic elements that make scripts easier to read and edit by human users. This semi-declarative behaviour places initialisation scripts among the DSL family of small programming languages [41], but markup languages, such as XML are also frequently used in this role.

ST2 – 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. The hard-coded equivalent for this type of scripting system can resemble the *state-effect pattern* [42], [18], which is frequently used in game development, as this category of script also includes rule-based scripting systems which can be used for the definition of domain knowledge in expert systems. An example for this are the intelligent NPCs (Non Player Characters), i.e. the computer controlled entities, that can be found in many computer games. Commercial computer games that use this kind of scripting system are Bioware's Role-Playing Games 'Neverwinter Nights' (Fig. 2) and 'Baldur's Gate'. $ST2$ type scripts are also frequently used to set up user interfaces (Fig. 6) and associated event handlers. Among the event based scripting systems there are two distinct sub-types:

ST2a – Event Handler Scripts:

The simpler sub-type of scripting systems in this category uses events that are built into its host game engine as pre-defined events. Here scripts only define the event handlers, i.e. the instructions that will be carried out when an event occurs. Scripts of this type may define additional conditions that influence the host application's event trigger mechanism. Events are triggered and event handlers are called from the game engine itself, when the events occur.

ST2b – Event Oriented Scripts:

The second sub-type are more sophisticated scripting systems that follow the concept of "Action Languages" as described by Gelfond and Lifschitz [43]. Scripts expressed in declarative languages, such as SGL [44], also fall into this category. These scripts first define the triggers and the situations in which these should act on events, i.e. the execution conditions, in addition to the event handlers themselves. In application implementations these trigger-definitions will usually be executed during the initialisation of the scripting system so that these events can be generated by the host game engine if all necessary preconditions are met. Once per execution cycle of the script, which in many games will happen once every frame, the conditions for triggering events will be checked against the current game-state, i.e. the in-game situation, and if these conditions evaluate as true they will induce the execution of the event handler. The examination of the game-state can happen through active polling of event data from the host game engine. Alternatively events can be triggered from within other events or posted as messages to the scripting system by the host game engine.

ST3 – Scripts which Run Like a Traditional Computer Program:

Finally there are the scripting systems that are modelled on "traditional" procedural, functional or object oriented programming languages that would immediately appear familiar to most programmers. Here we can identify two sub-types:

ST3a – Looping Scripts:

$ST3a$ scripts will be executed repeatedly to (re-)evaluate the current situation within the game, i.e. they will restart execution from the beginning of the script, once the end of the script has been reached. Effectively, scripts of this type are used to describe a single (high-frequency) control loop. They can be a superset of type $ST2$ scripts in the sense that they, too, are usable for expressing the *state-effect pattern* [42], [18], which is useful for continuously reevaluating current game states, such as those that manipulate state machines. If run once only at program start-up, scripts of this type are also suitable for use in similar environments as scripts of type $ST1$.

ST3b – REGULAR SCRIPTS:

Scripts of this type will execute once only, i.e. they will run from start to finish, concurrently with the host application. Consequently 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 ZBL/0 [45], which executes concurrent scripts for controlling NPCs in game environments.

A. Language Categorisation

A major difficulty in the categorisation of scripting systems is that many scripting languages can map onto more than one of the script categories, as their syntactic features can allow different modes of usage. For example, the popular scripting language Lua [46] would allow for the creation of scripting systems for each of the script types described here. Multi-language development [47], i.e. the combination of several programming and scripting languages within the same application, is a common feature of modern game development. A good example for this is the use of both XML and Python scripts within the game 'Sid Meier's Civilization IV' [48], in which the XML scripts for the provision of game settings are of type $ST1$, with higher level game functionality controllable through Python with type $ST2$ scripts. Multi-language development is also evident in 'The Meadow' [49], which employs Lua scrips both as type $ST1$ scripts for scene setup and as $ST2b$ scripts for controlling the system's graphical user interface (Fig. 6), while type $ST3b$ scripts written in the proprietary language C-Sheep are used to control the behaviour of virtual entities in the game world. The use of different script types that are both implemented using Lua in this example shows how easily the lines can be blurred and that the distinction of scripts needs to be made by mode of usage rather than by scripting language used.

Different criteria could be used for categorising scripts, i.e. instead of mode of usage, one could possibly categorise scripts by application domain, such as scene description languages – a term that could apply to many file formats for storing 3D geometry game assets – or behaviour definition languages [50] for defining NPC behaviour, which themselves would be a subcategory of artificial intelligence languages [51]. This, however, would lead to the placement of very similar types of scripts into different categories. As this might lead to confusion, we believe that our choice of categories by mode of usage is the more favourable alternative.

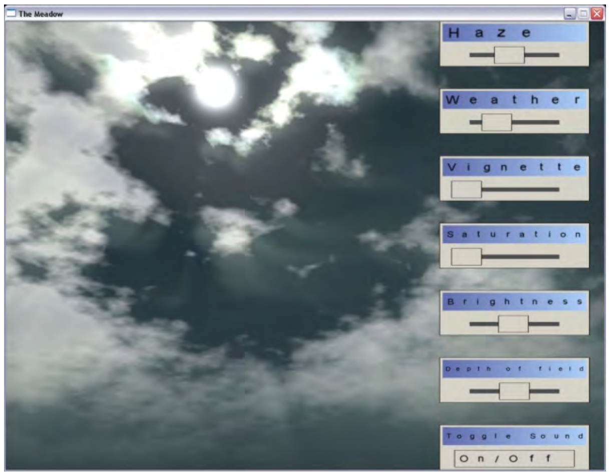


Fig. 6. Lua scripts of type $ST2b$ controlling the graphical user interface in 'The Meadow' [49] computer science education serious game.

V. CONCLUSIONS AND FUTURE WORK

Data-driven design takes program modularisation and code-reusability to its extremes and we have now reached a point in the trend towards data-driven development of computer games, where considerable sections of computer games are no longer hard-coded into the game programs themselves but instead are loaded in as scripted game assets. In this paper we have proposed a simple classification system for these scripts, categorising different types of script depending on their mode of usage, i.e. the manner in which they are executed within their host application.

"One characterization of progress in programming languages and tools has been regular increases in abstraction level – or the conceptual size of software designers building blocks" [52]. This observation is reflected in the success of domain specific scripting languages that provide bigger "building blocks" for specific operations that could be decomposed into simpler instructions, which would achieve the same overall effect but require a lot more effort by the application developer and result in the creation of a lot more source code. The higher level of abstraction provided by scripting systems shows them as a powerful alternative to the so-called implementation programming languages as they greatly reduce the effort required for giving complex instructions. In terms of scripting in computer games there is also an observable trend towards multi-language development [47]. Here the combined use of different languages and script types within the same host application greatly enhances the accessibility and usability of different features of the system, as scripts can be customised for the different domains that they are used in.

As scripting systems and the use of scripts in game development keep evolving, our classification of scripts may need to be revised and expanded in the future. An early indicator for this may be the emergence of concurrent systems and distributed applications, such as games created with the Network Scripting Language [53], which features scripts that, although similar to $ST3b$ type scripts, do not really map to any of our identified script categories.

One important area for future development lies in the area of development and content creation tools [54], which could be considered an integral part of a data-driven game engine [15]. Especially the continuing integration of scripting systems into game engines has prompted the need for tools to aid in the development of scripts. Although the high level of abstraction and frequently simple syntax of domain specific scripting languages usually allows non-programmers to cope perfectly with writing scripted programs for these systems, a popular approach to simplifying script generation for designers is the use of tools with a graphical user interface (GUI). These tools can range from simple text editors that have been extended to provide syntax highlighting for the scripting language (Fig. 2) to complex CASE (computer aided software engineering) applications that provide a more intuitive design approach to the generation of scripts. Some of these script manipulation tools are themselves written as scripts (of type $ST3b$), such as the 'Unit Formation Editor' in 'Civilization IV' [48], which was programmed in Python. However, most of the existing tools are mainly designed for the creation of scripts of types $ST1$ and $ST2$, which lack the complexity of type $ST3$ scripts that usually require a much greater understanding of programming, which in turn complicates the development of intuitive tools for their creation, which future work will need to address.

Acknowledgments

This paper includes imagery generated using the Virtual Egyptian Temple, which is a product of PublicVR (http://publicvr.org).

References

[1] I. Schramm-Wölk, "E-Learning und Edutainment," in *Proceedings of zfxCON05 - 2nd Conference on Game Development*, 2005, pp. 29–37.

[2] M. Zyda, "From visual simulation to virtual reality to games," *IEEE Computer*, vol. 38, no. 9, pp. 25–32, 2005.

[3] E. F. Anderson, L. McLoughlin, F. Liarokapis, C. Peters, P. Petridis, and S. de Freitas, "Serious games in cultural heritage," in *VAST 2009: 10th International Symposium on Virtual Reality, Archaeology and Cultural Heritage - VAST-STAR, Short and Project Proceedings*, 2009, pp. 29–48.

[4] B. Wilcox, "Reflections on Building Three Scripting Languages," Available from: http://www.gamasutra.com, 2007, [Accessed 01/11/2010].

[5] M. DeLoura, "The engine survey: Technology results," Gamasutra Expert Blogs - available from: http://www.gamasutra.com/blogs/MarkDeLoura/20090316/903/The_Engine_Survey_Technology_Results.php, 2009, [Accessed 01/11/2010].

[6] B. Campbell, "Swiss Army Chainsaw: A Common Sense Approach to Tool Development," Available from: http://www.gamasutra.com, 2006, [Accessed 01/11/2010].

[7] D. Given, "The inComplete SCUMM Reference Guide," Available from: http://www.scummvm.org, 2002, [Accessed 29/02/2008].

[8] B. Kane, "SIGGRAPH: EA's Entis on Derailing the 'Commoditization Treadmill'," Gamasutra Industry News, available from: http://www.gamasutra.com, 2007, [Accessed 01/11/2010].

[9] J. Noghani, F. Liarokapis, and E. F. Anderson, "Randomly generated 3d environments for serious games," in *VS-GAMES 2010: Proceedings of the 2nd International Conference on Games and Virtual Worlds for Serious Applications*, 2010, pp. 3–10.

[10] A. BinSubaih, S. Maddock, and D. Romano, "A Survey of 'Game' Portability," University of Sheffield, Tech. Rep. CS-07-05, 2007, department of Computer Science.

[11] J. Jacobson and L. Holden, "The virtual egyptian temple," in *ED-MEDIA: Proccedings of the World Conference on Educational Media, Hypermedia & Telecommunications*, 2005.

[12] J. Jacobson and M. Lewis, "Game Engine Virtual Reality with CaveUT," *IEEE Computer*, vol. 38, no. 4, pp. 79–82, 2005.

[13] J. Jacobson, K. Handron, and L. Holden, "Narrative and content combine in a learning game for virtual heritage," in *Computer Applications to Archaeology 2009*, 2009.

[14] S. Rabin, "The Magic of Data-Driven Design," in *Game Programming Gems*. Charles River Media, 2000, pp. 3–7.

[15] E. F. Anderson, S. Engel, L. McLoughlin, and P. Comninos, "The case for research in game engine architecture," in *Proceedings of the ACM FuturePlay 2008 International Academic Conference on the Future of Game Design and Technology*, 2008, pp. 228–231.

[16] Danc, "Managing Game Design Risk: Part II – Data Driven Development," Blog Entry, available from: http://www.lostgarden.com, 2006, [Accessed 01/11/2010].

[17] K. Wilson, "Data-Driven Design," Blog entry, available from: http://www.GameArchitect.net, May 2002, [Accessed 01/11/2010].

[18] W. White, C. Koch, J. Gehrke, and A. Demers, "Better scripts, better games," *Commun. ACM*, vol. 52, no. 3, pp. 42–47, 2009.

[19] A. Wallis, "Is Modding Useful?" in *Game Carreer Guide 2007*. CMP Media, 2007, pp. 25–28.

[20] A. Varanese, *Game Scripting Mastery*. Premier Press, 2003.

[21] OUP, "Scripting Language," *A Dictionary of Computing*. Oxford University Press, 2002.

[22] P. Sweetser and J. Wiles, "Scripting versus Emergence: Issues for Game Developers and Players in Game Environment Design," *International Journal of Intelligent Games and Simulations*, vol. 4, no. 1, pp. 1–9, 2005.

[23] J. K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century," *IEEE Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[24] B. W. Kerninghan and C. J. Van Wyk, "Timing Trials, or the Trials of Timing: Experiments with Scripting and User-Interface Languages," *Software: Practice & Experience*, vol. 28, no. 8, pp. 819–843, 1998.

[25] B. Dawson, "Game Scripting in Python," in *Proceedings of the 2002 Game Developers Conference*, 2002.

[26] N. Bezroukov, "Scripting Languages as a Step in Evolution of Very High Level Languages," 2006, available from: http://www.softpanorama.org/People/Scripting_giants/scripting_languages_as_vhll.shtml [Accessed 01/11/2010].

[27] D. G. Korn, "ksh - An Extensible High Level Language," in *Very High Level Languages Symposium (VHLL)*, 1994, pp. 129–146.

[28] A. V. Aho, B. W. Kernighan, and P. J. Weinberger, "Awk - a Pattern Scanning and Processing Language (Second Edition)," *Software: Practice & Experience*, vol. 9, no. 4, pp. 267–280, 1979.

[29] D. Gould, *Complete Maya Programming: An Extensive Guide to MEL and the C++ API*. Morgan Kaufmann, 2002.

[30] M. Harmon, "Building Lua into Games," in *Game Programming Gems 5*. Charles River Media, 2005, pp. 115–128.

[31] M. West, "Domain-Specific Languages," *Game Developer*, vol. 14, no. 7, pp. 33–36, 2007.

[32] S. Rabin, "Implementing a State Machine Language," in *AI Game Programming Wisdom*. Charles River Media, 2002, pp. 314–320.

[33] B. W. Kernighan and R. Pike, *Using Macros to Generate Code*. Addison-Wesley, 1999, ch. 9.6.

[34] S. Rabin, "Finding Redeeming Value in C-Style Macros," in *Game Programming Gems 3*. Charles River Media, 2002, pp. 26–37.

[35] L. Prechelt, "Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java," *Advances in Computers*, vol. 57, pp. 205–270, 2003.

[36] D. Garcés, "Scripting Language Survey," in *Game Programming Gems 6*. Charles River Media, 2006, pp. 323–340.

[37] S. A. Baba, H. Hussain, and Z. C. Embi, "An Overview of Parameters of Game Engine," *IEEE Multidisciplinary Engineering Education Magazine*, vol. 2, no. 3, pp. 10–12, 2007.

Published in Proceedings of VS-GAMES 2011: 3rd International Conference on Games and Virtual Worlds for Serious Applications, pages 47-54; definitive version available from http://dx.doi.org/10.1109/VS-GAMES.2011.13

[38] R. Huebner, "Adding Languages to Game Engines," *Game Developer*, vol. 4, no. 9, 1997.

[39] P. Tapper, "Personality Parameters: Flexibly and Extensibly Providing a Variety of AI Opponents' Behaviors," Available from: http://www.gamasutra.com, 2003, [Accessed 01/11/2010].

[40] A. Sherrod, *Ultimate 3D Game Engine Design & Architecture*. Charles River Media, 2007.

[41] A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[42] W. White, B. Sowell, J. Gehrke, and A. Demers, "Declarative processing for computer games," in *Sandbox '08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, 2008, pp. 23–30.

[43] M. Gelfond and V. Lifschitz, "Action Languages," *Linköping Electronic Articles in Computer and Information Science*, vol. 3, no. 16, 1998.

[44] R. Albright, A. Demers, J. Gehrke, N. Gupta, H. Lee, R. Keilty, G. Sadowski, B. Sowell, and W. White, "SGL: a scalable language for data-driven games," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1217–1222.

[45] E. F. Anderson, "A NPC Behaviour Definition System for Use by Programmers and Designers," in *Proceedings of CGAIDE 2004 5th Game-On International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004, pp. 203–207.

[46] R. Ierusalemschy, L. H. de Figueiredo, and W. Celes, "The Evolution of Lua," in *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007, pp. 2–1–2–26.

[47] A. M. Phelps and D. M. Parks, "Fun and games: Multi-language development," *Queue*, vol. 1, no. 10, pp. 46–56, 2004.

[48] M. Thamer, *Sid Meier's Civilization IV Tech & Modding Overview*, 2005, available from: http://www.cyberstump.com/civ4/docs/civ4_features/tech_and_modding_overview.doc [Accessed 01/11/2010].

[49] E. F. Anderson and L. McLoughlin, "Critters In The Classroom: A 3D Computer-Game-Like Tool for Teaching Programming to Computer Animation Students," in *ACM SIGGRAPH 2007 Educators Program*, 2007.

[50] E. F. Anderson, "Scripted smarts in an intelligent virtual environment: behaviour definition using a simple entity annotation language," in *Future Play '08: Proceedings of the 2008 Conference on Future Play*, 2008, pp. 185–188.

[51] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan, "Scaling games to epic proportions," in *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 31–42.

[52] D. Garlan and M. Shaw, "An Introduction to Software Architecture," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-94-TR-21, ESC-TR-94-21, 1994, CMU Software Engineering Institute.

[53] G. Russell, A. F. Donaldson, and P. Sheppard, "Tackling online game development problems with a novel network scripting language," in *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, 2008, pp. 85–90.

[54] J. Blow, "Game development: Harder than you think," *Queue*, vol. 1, no. 10, pp. 28–37, 2004.