Architectural studies of games engines — The quake series

Conference Paper · September 2009

DOI: 10.1109/ICEGIC.2009.5293600 · Source: IEEE Xplore

Cornelia Boldyreff
University of Greenwich
111 PUBLICATIONS **1,059** CITATIONS
SEE PROFILE

Andrea Capiluppi
University of Groningen
112 PUBLICATIONS **1,366** CITATIONS
SEE PROFILE

Architectural Studies of Games Engines – the Quake Series

James Munro[*], Cornelia Boldyreff[**], and Andrea Capiluppi[**]

[*]Centre of Research on Open Source Software, Department of Computing, University of Lincoln, UK,
`jmunro@students.lincoln.ac.uk`,
[**]School of Computing, Information Technology and Engineering, University of East London, UK,
`{a.capiluppi, c.boldyreff}@uel.ac.uk`

Abstract

The move of commercial companies to "open-source" their products presents challenges for both the proposing company and the wider open source (OS) communities. The former has to align their source code to the OS practices, while the latter has to cope with large amounts of closely-developed code.

This paper aims to present relevant data and results from the analysis performed on the Quake family of OSS game engines, including findings and an initial interpretation of the data. This forms the basis for the architectural understanding necessary to design and develop improvements and new features to the studied game engines.

It constitutes a major resource for games developers who wish to contribute to the further evolution of these games engines; and it provides insights into how the Quake engine architecture has evolved in practice since it was released as an open source project.

1 Introduction

Game engine development is typically performed in a closed environment overseen by the company leading the project. These engines are often licensed to other companies as middleware for use in their own games [5]. In recent years, game engines that have been developed in such a manner have been released to the public as open-source projects [15]. This gives a unique opportunity of comparison. On the one hand, a mixed-style software project can be studied, where the "closed-source" phase was followed by its release as open source, and new developers are attracted to work on the project. On the other hand, "traditional" open source projects provide large amounts of data; patterns of organisation of work in the latter can be compared with the fully-closed, or the mixed closed-open using quantitative and qualitative analyses.

Analysis of the architecture and structural composition of these engines could provide a valuable insight into game engine design and development. In order for this research to be credible it is important to support it with key software engineering considerations with particular regard to software architecture, evolution and maintenance. The importance of a system's architecture is well-known as it can affect many factors such as performance and maintainability (Bosch, 2000, cited in [16]); both are critical aspects of game engines. As any game player knows, performance is an important factor as game-play can often be affected by faltering performance of a game engine.

A study of the evolution of a particular series of game engines, specifically the 'Quake' series, may identify structural flaws and common pitfalls that could be avoided in future developments. This may correlate directly with Lehman's laws of program evolution, it is clear even at this early stage that the Quake series of engines is an example of Lehman's first law of 'Continuing Change', which proposes that a program must continually adapt or become progressively less satisfactory [11]. The Quake series is a prime example of this because three distinctly unique iterations of the engine, I, II and III are available as open-source projects [10], each a natural evolution of the former with increasing complexity. This itself is underlined by Lehman's second law, 'Increasing Complexity' that states that regressions in the program's architecture can be expected unless the system is adequately maintained as it evolves. By taking other game engines into account it may be possible to identify a common architecture existing between separately developed projects; and methods of unifying the approach to game engine design may be evident.

Little research currently exists specifically in direct relation to game engine architecture; however, common architectures have previously been identified in software engineering studies based on different categories of open-source software; in particular, instant-messaging (IM) applications by [3]. By replicating aspects of Knowles' and Capiluppi's study of popular open-source IM applications it may be possible to generate similar findings with regards to game engines. A basic analysis of the architectural composition of a game engine may be performed using well-known tools such as Doxygen, a source-code documentation generator.

Apart from stripping all the comments from the source files of a project, Doxygen extracts also the dependencies and the method invocations (and function calls) of the elements composing the source code. When aggregated at the project level, this can provide a structural overview of a project's source-code [18], helping to identify the existence of individual components within the project. At a deeper level of analysis, manual code observation can be carried out to gain a thorough understanding of specific functionality, providing detail unattainable through the use of automated tools; however, this technique should be used sparingly due to its time-consuming nature. In this paper, a mixed approach will be used: automatic extraction of code dependencies will be followed by a manual analysis of the elements that could be contained within the architectural components.

Finally, certain tools exist that can produce visual representations from specific data in the form of graphs. In particular, Graphviz will be used to generate graphs that may help to clarify the architecture of an engine in a visual manner [7].

2 Selected Game Engines

Table 1 presents the game engines that have been selected for analysis and examination. Initially, various other engines were considered for analysis but these were later discarded in favour of focusing purely on the Quake-series of games. This choice has allowed the study to focus on the evolution of this family of games and their engines over the course of their lifetime.

Table 2 highlights key information regarding the source-code composition of the engines. It is important to note that these engines were initially developed as commercial game engines and were later released to the general public years after their mainstream success. This essentially means that the engines do not reflect traditional open-source project characteristics with the exception of the *ioquake3* fork that has seen continued community development to the *id Tech 3* engine after its public release.

Game	*Engine*	*Available From*
Quake I	Quake 1	`http://tinyurl.com/43wloy`
QuakeWorld	QuakeWorld	`http://tinyurl.com/43wloy`
Quake II	id Tech 2	`http://tinyurl.com/3feehm`
Quake III	id Tech 3	`http://tinyurl.com/cha9q`
ioquake3	id Tech 3+	`http://tinyurl.com/c5rjyb`

Table 1: Selected game engines.

The type of license for each engine has been determined by viewing the appropriate text file within each project directory. The *Organised* column represents whether the project is sorted into folders and sub-folders representing different modules of functionality (see section 4.1 below for more details). In traditional OSS projects, it has been noted that folder structure (or *physical* architecture, [2]) plays a relevant role for the parallel work of distributed developers in a software release [17]. The objective of the next subsection will be to understand whether a similar approach is used by commercial organisations when developing their code. Finaly, the date for *ioquake3* has been obtained by examining the public Subversion (SVN) logs [9].

Engine	*Lang*	*Organised*	*License*	*Release Date*
Quake I	C	No	GPL v2	21/12/1999
QuakeWorld	C	No	GPL v2	21/12/1999
id Tech 2	C	Yes	GPL v2	21/12/2001
id Tech 3	C	Yes	GPL v2	19/08/2005
ioquake3	C	Yes	GPL v2	26/08/2005

Table 2: Summary characteristics of selected game engines.

3 Characterisation Analysis

This section details the analytical steps to provide initial analysis results and insights into the evolution of the Quake-series of game engines.

3.1 SLOC Analysis

A brief analysis using the SLOCCount [19] tool can give an insight to the size of a software system in terms of pure source-code. This alone is useful to assess the complexity of a system, but is also useful to see how the size of a system has evolved over time. It should not be assumed that a system will only increase in size and complexity over its lifetime. There are methods of corrective maintenance and preventative maintenance that can reduce the size and complexity of a system through code refactoring [20]. This is important because as pointed out, the less code in a game engine the more efficiently it may perform [1].

3.1.1 Quake-series in Comparison

A SLOC count comparison of all the selected game engines demonstrates an increase in complexity with each version. The count between major versions highlights an almost double increase each time. Minor versions of the engine such as *QuakeWorld* and *ioquake3* show less of an increase as expected. The *ioquake3* engine particularly is described as a bug-fix release of *id Tech 3*, demonstrating corrective maintenance at the hands of the open-source community.

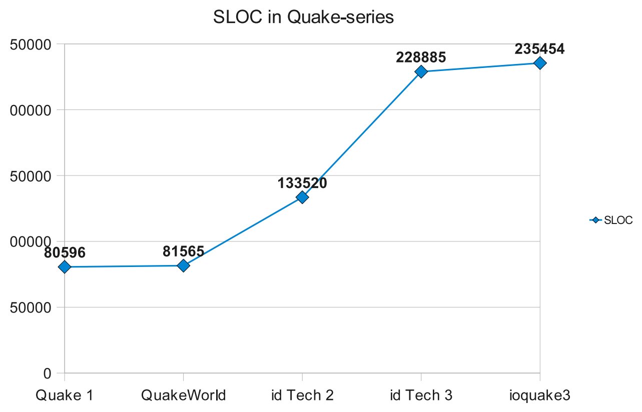

Figure 1: An increase in SLOC count is consistently visible across the Quake-series.

3.1.2 Language Composition of the Quake-series

A study of the composition of source files was performed for each of the engine versions, and the results are displayed in Table 3. The following observations can be drawn:

	id Tech 1	Quake-World	id Tech 2	id Tech 3	ioquake3
ansic	72,542	63,799	122,680	267,449	234,442
asm	7,745	17,587	8,330	1,362	1,080
cpp				59,242	
objc			2,458	6,563	
perl				6,320	
sh	309	179	52	375	720
yacc					185

Table 3: SLOC in *Quake* series – by Language

- The most common programming language used in *Quake 1* is C by a broad margin. C is considered a fast and efficient programming language and C++ did not become common in game development until recently in the late nineties [14].
- The C programming language is clearly identifiable as the most prominent. The presence of other languages can possibly be attributed to the inclusion of the source-code for tools in the project folder.
- There is a higher proportion of ASM in *id Tech 2* than prior releases of the engine. This is most likely used in performance-intensive sections of the engine such as complex mathematical calculations that are crucial to the engine's efficiency. As hardware performance has increased over the years, the use of ASM in performance intensive sections of game engines has reduced.
- The official *id Tech 3* distribution consists of a wide selection of programming languages. This is mostly due to the inclusion of stand-alone tools that are used to create content for the game engine itself. The main engine itself still predominantly consists of the C programming language.
- A reduction in different programming languages in *ioquake3* is most likely due to the project not containing many of the tools that are distributed with the official *id Tech 3* release. A brief manual examination of the project's source folder confirms that tools are missing, such as the level editing tool *q3radiant*.

4 Extracting the Architecture

There are multiple ways of extracting the underlying architecture from an existing software system. Different methods produce different results and so a combination approach is often beneficial. There is a certain amount of investigative work required to assess the architecture of a software system, especially large systems such as 3D game engines. The following subsections detail the processes used to examine the architecture of the selected game engines, a difficult task when considering the sheer size of the engines in question (*id Tech 3* contains approximately 250,000 SLOC, see Section 3.1 below).

For each of the selected engines, notable modules and their purpose are identified from the contents of their source-code by analysing the following data obtained through the Doxygen tool:

1. Function names and dependencies,
2. Source-code file names and dependencies, and
3. Original developer source-code comments,

4.1 Architecture from Folder Hierarchy

A simple method of determining the overall architecture of a system is to identify the individual components that make up the system as a whole. By examining the system's project organisation and folder hierarchy it is possible to identify portions of source code that have been grouped into folders, usually by the nature of their functionality, a common trait of software projects [12]. This has been reported in the OS literature as "state of the art": when evolving to large sizes, OS projects tend to organise their source code into semantically-rich folders, in order to ease the distributed effort of OS developers [17]. This is a similar approach to the one used in [3] in their analysis of IM application architectures. There is a tendency for newer projects to have a disorganised folder hierarchy but this improves as a project reaches maturity through project restructuring, as highlighted by [3].

The assumption that software projects always separate system components into individual folders of specific functionality, was challenged in practice; in the first studied versions of this system (*Quake 1*), it is evident that it did not separate modules into individual folders of functionality. The objective in the second part of the paper will be to investigate whether this pattern has evolved with the other studied releases of the game engine.

Since *Quake 1* does not provide semantically-rich folder names, or a proper folder structure, a different technique had to be used to identify the relevant engine modules. On a manual inspection, the source-code files are prefixed with letters that easily identify their membership to a particular module. For example, source-code files that contribute to the networking functionality of the engine are prefixed with *net_*, making these files easy to distinguish.

The next section will present the abstract architecture of the *Quake 1* engine, extracted through folder hierarchy analysis and examination of documentation reports produced by the Doxygen tool.

4.2 Architecture of Quake 1

In *Quake 1* the main modules are not organised into separate folders, and the process of identifying modules increases in difficulty. The main project folders here are used to encapsulate functionality from third-party libraries and tools such as *dxsdk* (DirectX SDK) and *gas2masm*.

4.2.1 Notable Modules

Due to the lack of folder organisation, the key to identifying the notable modules listed below involved examining the data extracted using the Doxygen tool. Specifically, function prefixes and source-file names have been used to identify which modules of functionality code belongs to. For example, source-file names beginning with the *net_* prefix can easily be identified as belonging to the network-code functionality.

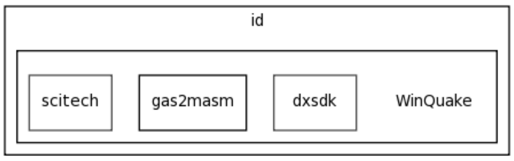

Figure 2: The architecture of *Quake 1* from a folder-hierarchy perspective.

Module	*Functionality*
client	Responsible for the client-side game code. This is the client component of the distributed client/server architecture model.
game	Responsible for the server-side game code. This is the server component of the distributed client/server architecture model.
vid	Handles the graphics rendering functionality. Similar to the *render* modules found in subsequent versions of the engine.
sys	Handles platform specific issues such as independent code for both the MS-DOS and Microsoft Windows platforms.
snd	Audio playback functionality for client-side sound effects and music.
net	Networking functionality, responsible for communication between *client* and *server* modules.

Table 4: Notable modules of Quake 1

Table 4 demonstrates the key modules extracted using the Doxygen-aided process. The *Module* column represents the name of the module functionality extracted from the source-code. Fortunately, they are named logically, semantically and phonetically which makes the process of identifying them easier.

4.3 Architecture of QuakeWorld

This section presents the physical architecture of the *QuakeWorld* engine, extracted through folder hierarchy analysis and examination of documentation reports produced by the Doxygen tool. Though *QuakeWorld* contains a more organised structure than its predecessor *Quake 1*, there is still little information to be obtained from hierarchy analysis. In order to identify an increased amount of detail the same process of examining source-file and function name prefixes was used.

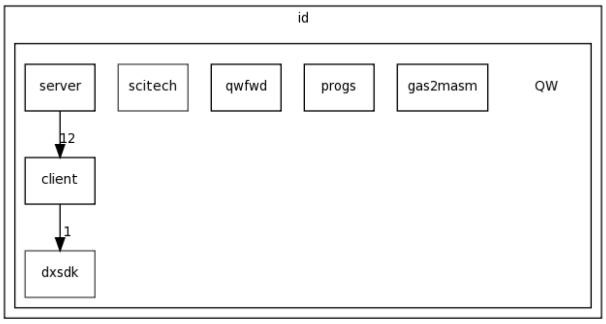

Figure 3: The architecture of *QuakeWorld* from a folder-hierarchy perspective.

4.3.1 Notable Modules

The key modules identified in *QuakeWorld* remain largely unchanged from an abstract perspective. The inner functionality within these modules may have altered but the same general structure remains.

The most promising result of this analysis is the inclusion of separate sub-folders to encapsulate the *client* and *game* modules (see Figure 3). This may have a direct link to the fact that *QuakeWorld* was an incremental improvement of the original engine designed to improve the networking functionality for multi-player purposes. This could be interpreted as an attempt to separate the client functionality from the game-logic contained in the *game* module. The arrows seen in the Figure represent folder-to-folder references (method invocations, function calls or file dependencies).

4.4 Architecture of id Tech 2

This section presents the abstract architecture of the *id Tech 2* engine, extracted through folder hierarchy analysis and examination of documentation reports produced by the Doxygen tool.

This version of the Quake engine represents the first version to contain some order of folder organisation: it could be argued that one of the effects of “open-sourcing” a commmercial application has the effect of reorganising the source code in semantically-rich source folders. As a result of this, the folder-hierarchy figure demonstrates an apparently more complex architecture. As seen above, the arrows seen in Figure 4 represent folder-to-folder references, the number indicates the frequency of these references.

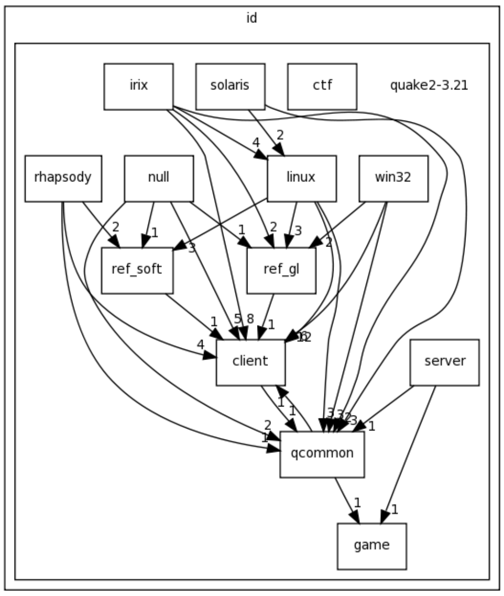

Figure 4: The architecture of *id Tech 2* from a folder-hierarchy perspective.

Figure 4 demonstrates an interesting separation of functionality between the *client* and *server* modules, representing a clear division in purpose. Both of these modules appear to ’consume’ the *game* module which provides the game-logic code. From this it can be assumed that the game-logic has been written in a reusable way that provides suitable abstraction that allows it to be used without prior knowledge of the ’consuming’ module.

4.4.1 Notable Modules

These key modules were identified using a combination of the previously discussed process and by examining the chart produced automatically by Doxygen using GraphViz.

Platform-specific code modules such as *solaris*, *linux* and *win32* have been omitted from the analysis as they provide only simple functionality designed to load the rest of the game engine onto specific platforms. These modules were identifiable by name and also their relationship to the *client* module that they are responsible for loading.

4.5 Architecture of id Tech 3

This section presents the abstract architecture of the *id Tech 3* engine, extracted through folder hierarchy analysis and examination of documentation reports produced by the Doxygen tool.

The architecture of the engine is becoming increasing complex with each incremental version in the series, this can be clearly seen by comparing Figure 4 and Figure 5.

Module	*Functionality*
client	Responsible for the client-side game code. This is the client component of the distributed client/server architecture model.
game	Responsible for the server-side game code. This is the server component of the distributed client/server architecture model.
qcommon	Common functionality and data structures used throughout the engine.
ref_gl	Responsible for rendering the client-side graphics rendering using hardware acceleration in-conjunction with the OpenGL graphics library.
ref_soft	Responsible for the client-side graphics rendering by implementing a software accelerated rendering module.

Table 5: Notable modules of id Tech 2

The increase in the amount of individual folders has also introduced a more complex relationship between modules.

4.5.1 Notable Modules

These key modules were identified using a combination of the previously discussed process and by examining the chart produced automatically by Doxygen using GraphViz.

Module	*Functionality*
botlib	Functionality responsible for controlling computer controlled game players using artificial intelligence techniques.
bspc	Description.
cgame	Client-side game code, similar to previous engine versions.
client	Main client executable. Loads the client-side game module and can be used to connect to a game server, either locally or remotely hosted.
game	Server-side game code, similar to previous engine versions.
q3_ui	User-interface module, provides a visual interface for players. Client-side only.
renderer	Graphics rendering library, similar to other engine versions. Implemented as a single module rather than two individual modules as seen in *id Tech 2*.
server	Main server executable. Loads the the main server-side code and provides services to game clients.

Table 6: Notable modules of id Tech 3

The increase in the total number of modules of functionality may be attributed to the clearer abstraction of individual types of functionality. For example, the *botlib* module functionality may have been present in previous versions (though likely in a simpler form) but had been simply included in the main application module, rather than making a clear distinction between types of functionality.

4.6 Architecture of ioquake3

This section presents the abstract architecture of the *ioquake3* engine, extracted through folder hierarchy analysis and examination of documentation reports produced by the Doxygen tool.

The architecture of *ioquake3* remains largely unchanged from *id Tech 3*. The project sees some organisational changes in terms of its directory structure and different third-party libraries have been introduced to provide features and functionality that did not exist in the parent project.

Due to the small quantity of differences between *id Tech 3* and *ioquake3* there is no need to elaborate on the notable modules as they remain unchanged from the previous version.

4.7 Cloning and Code Reuse in the Quake-series

As the final attempt to characterise the source code of the Quake series, it was studied the degree of “cloning” between subsequent versions of the engine. The scope of this analysis does not concern the existence of code clones within the Quake-series of engines on an individual basis; however, the process of detecting clones can be adapted to provide a summarised calculation of the amount of shared code between different versions of the engine as a device for understanding the evolution of the series.

4.7.1 Code Clones

Code clones are generally considered as bad artifacts of a software system [13]. They usually occur when a programmer reuses a portion of existing code that is known to serve a particular function. Across different software projects this is more acceptable, but within the same project this creates redundancy of code and can cause serious maintenance issues. If the duplicated code contains a bug or exploit, every single instance of it within the project must be located and fixed. If a situation arises where code is reusable within a system, a better solution is to turn it into a reusable function that will then be the only portion that needs fixing in the event of a bug.

4.7.2 Process

To identify how much code was reused between different versions of the Quake engine, a code clone analysis tool

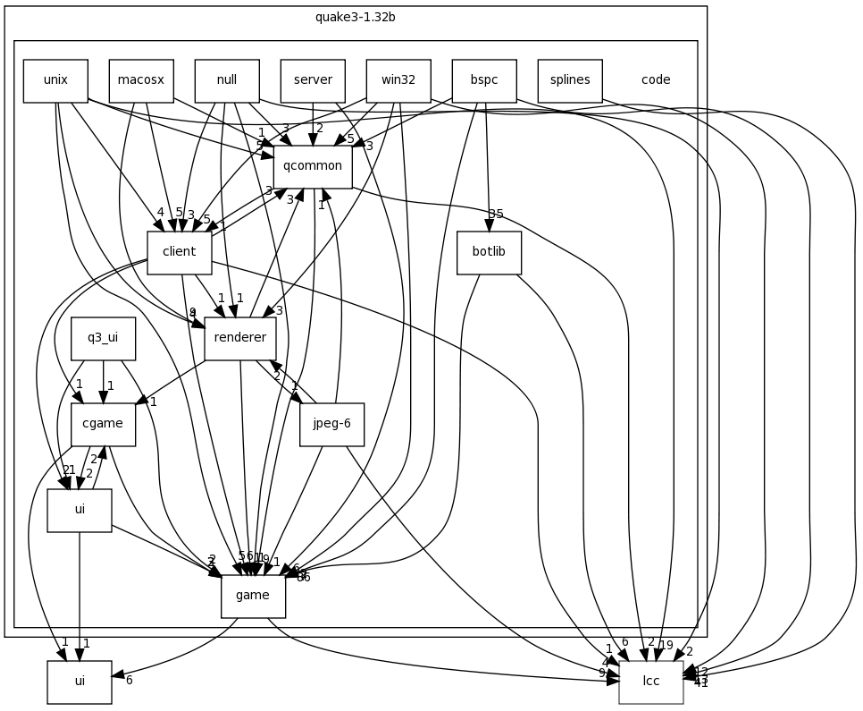

Figure 5: The architecture of *id Tech 3* from a folder-hierarchy perspective.

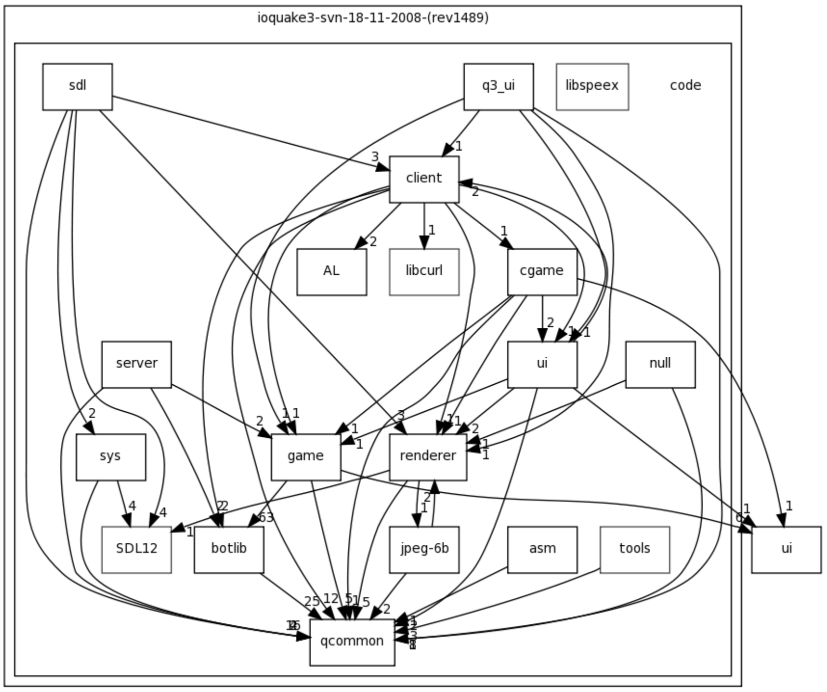

Figure 6: The architecture of *ioquake3* from a folder-hierarchy perspective.

was used. There are various tools that fit this purpose, but for convenience and ease-of-use Simian was used. Simian [8] is designed for analysing code clones within a single project, rather than duplicate code across multiple projects. To circumvent this issue, the following process has been devised:

- Analyse code clones within *Project A*.
- Analyse code clones within *Project B*.
- Combine *Project A* and *Project B* as two folders of a new project, *Project C*.
- Analyse code clones within *Project C*.
- From the total clones found in *Project C*, minus the sum of the clones found in *Project A* and *B*.
- The result of this is the number of code clones shared between two projects.

From this process the amount of shared code between two versions of the Quake engine can be quantified. Initially the accuracy of this method was unproven; however, after some manual checking of file-to-file differences using the GNU Diff [4] tool it became clear that this method could demonstrate the amount of shared code between selected versions of the Quake engine and provide a simple means to better understand its evolution.

It is likely that other more accurate tools and methods exist to measure the amount of shared code between projects, but for the scope of this study the devised process is sufficient.

4.7.3 Shared Code with Quake 1 Engine

Figure 7 below shows the amount of shared code between *Quake 1* and other engines in the Quake-series. With each major revision of the engine there is a reduction in the amount of shared code. *QuakeWorld* sees less of a reduction in shared code due to being a minor re-write of the original *Quake 1* engine designed to introduce better networking functionality.

The *ioquake3* engine also sees less of a reduction as it is based on *id Tech 3* and aims to provide a concise base for developers to base their own games on.

4.7.4 Reusable Candidates

A manual analysis of some of the detected shared code between different versions of the engine indicated that certain components are likely to be reused consistently. For example, the 3D mathematical calculation functions are generic to many 3D uses and are unlikely to change through different versions of the engine.

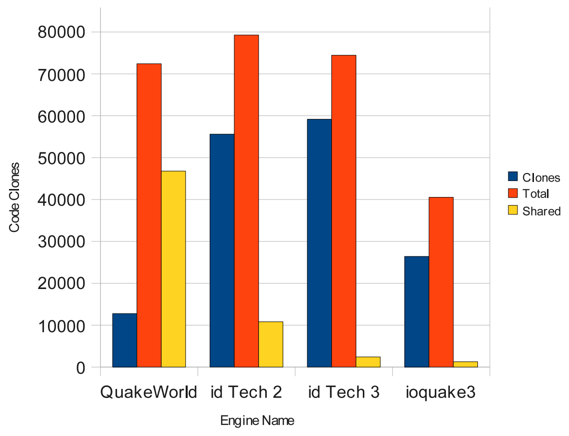

Figure 7: The amount of shared code between each version of the Quake-series and *Quake 1*.

4.7.5 id Tech 4 and Future Versions

The Quake-series sees continued development in a commercial environment to support new games and technologies. The next version of the engine known as *id Tech 4* has not yet been released to the public as an open-source project and so cannot be analysed as part of this study. It is a known fact that the engine has been completely redesigned using C++ with an OOP paradigm instead of C used in its predecessors and so it is highly unlikely that much if at all any shared code exists between *id Tech 4* and *Quake 1*.

4.8 Inter-Module Communication

Individual modules of functionality need efficient methods of communication between themselves. As previously discussed in this report, *virtual functions* are a popular means of facilitating this. The Quake-series of engines are implemented in the C programming language and do not make use of this language-specific feature.

In the Quake-series, communication is handled by passing *pointers* to C structures that exposes the functionality contained within a module. This method is efficient and incurs considerably less overhead than would be through the use of *virtual functions*. The Quake-series generally uses one clear structure per-module of functionality as an interface to other modules, employing smaller structures for internal purposes and data representation.

One disadvantage of this approach is that third-party applications or even modules can obtain pointers to these structures and essentially eavesdrop on the inter-module communication and exploit the data for their own uses. This is a popular method used to cheat in online games. To combat this, usually some form of third-party anti-cheat tool such as PunkBuster [6] is used to protect the engine's memory address space from external interference.

5 Limitations and Future Works

- more metrics with respect to the software architectures.
- based on runtime behavior and semantic code analysis;
- the comparison of performance between the different versions and the analysis of the relationships between performance and architecture choices.

6 Summary of Results, Conclusions and Further Research

This research has examined and identified the evolving architecture of the Quake-series of game engines. It has argued that, when released as an OS project, a software system has to adapt to the common practices of OS developers, one of which being the partinioning of the source code into various folders and subfolders, in an attempt to organise it.

The initial Quake engine is initially disorganised, reflecting the closed environment and internal practices of the company developing it; later, it becomes more and more organised in terms of its physical architecture. The engine has also increased in complexity with each release, along with a large increase in SLOC count in every major version. The broadening gap between the last entry in the series, *id Tech 3* and the first, *Quake 1* is easily distinguishable through the identification of rapidly decreasing quantities of shared code.

During the architectural analysis of the engines, common modules of functionality have been frequently discovered. The following list refers to these modules by a generic name interpreted from their original names (which differed slightly across different versions):

- Common functionality module.
- Game-logic module (abstracted from both client and server).
- Server specific functionality module.
- Client-side functionality module.
- Sound and video output module (separately split in later versions).

This common functionality specifically represents suitable candidates for the implementation as a reusable code library, potentially useful for different game engines that are unrelated to the Quake family due to the commonality of the functionality the reusable modules would encapsulate.

The results of the analysis described above have recently been used in the process of identifying suitable improvements and enhancements to a specific engine and have supported implementing these in an appropriate manner. Different improvements have been considered and justified in a feasibility study based on these results. The modifications are aimed at the *id Tech 2* engine, based on the results of the architectural analysis. The engine marks the midpoint in the series before the complexity of both architecture and size bloomed with the advent of *id Tech 3* and it is potentially easier to enhance. The potential for building a library of reusable code for games engines more generally will be explored as a topic of further research.

References

[1] A. T. (andy.thomason@snsys.com). Re: Advice for a university of lincoln games programming student. Email to James Munro (james@jamesdesign.org)., Monday 5th January 2009.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, December 1997.

[3] A. Capiluppi and T. Knowles. Software engineering in practice: Design and architectures of floss systems. In *5th International Conference on Open Source Systems (OSS 2009)*, 2009.

[4] P. Eggert. Diffutils - gnu project - free software foundation [online], 2008. Accessed Saturday 7th Febraury 2009.

[5] I. Epic Games. Unreal technology [online], 2008. Accessed Wednesday 5th November 2008.

[6] Even-Balance. Punkbuster online countermeasures [online], 2009. Accessed 8th February 2009.

[7] Graphviz. Graphviz [online], 2006. Accessed Sunday 2nd November 2008.

[8] S. Harris. Simian - similarity analyser — duplicate code detection for the enterprise [online], 2008.

[9] Icculus. [quake3] log of /trunk/code [online], 2005. Accessed Saturday 7th February 2009.

[10] id Software. id software: Technology downloads [online], 2008. Accessed Wednesday 5th November 2008.

[11] M. M. Lehman and L. A. Belady. Program evolution: Processes of software change. *A.P.I.C. Studies In Data Processing*, 27, 1985.

[12] N. C. Mendonca and J. Kramer. Component module classification for distributed software understanding. In *15th IEEE International Conference on Software Maintenance*, page Page 119, 1999.

[13] T. Mens and S. Demeyer. *Software Evolution*. Springer, 2008.

[14] A. Rollings and D. Morris. *Game Architecture and Design*. The Colriolis Group, Scottsdale, Arizona, 2000.

[15] Slashdot. Slashdot — quake 3: Arena source gpl'ed [online], 2005. Accessed Sunday 2nd November 2008.

[16] I. Sommerville. *Software Engineering*. International Computer Science Series. Addison-Wesley, 8th edition edition, 2007.

[17] D. Spinellis. *Code Reading: The Open Source Perspective*. Addison-Wesley Professional, May 2003.

[18] D. van Heesch. Doxygen [online], 2008. Accessed Sunday 2nd November 2008.

[19] D. A. Wheeler. Sloccount [online], 2008. Accessed Wednesday 10th December 2008.

[20] R. J. Wirfs-Brock. Enabling change. *IEEE Software*, Volume 25(Number 5):Pages 70–71, September / October 2008.