Image and Vision Computing New Zealand, Dunedin, New Zealand, 28 - 29 Nov, 2005.

A Framework for Game Engine Based Visualisations

Burkhard C. Wünsche, Blazej Kot, Andrew Gits, Robert Amor, John Hosking and John Grundy

University of Auckland, Dept. of Computer Science, Private Bag 92019, Auckland, New Zealand.
Email: burkhard@cs.auckland.ac.nz

Abstract

Game engines are the core software component of video games and typically handle tasks such as rendering, game AI and collision detection between game objects. Due to the popularity of computer games, a huge amount of research has been devoted towards the development of game engines. In this paper we analyse the suitability of game engines for visualisation research. We present a software architecture and a visualisation framework which facilitates this task and we evaluate the suitability of a number of popular engines. We conclude with a summary of our experiences from several case studies.

Keywords: game engines, visualisation, human-computer interfaces, collaborative interfaces

1 Introduction

Modern computer games make use of technologies from many areas of computer science: graphics, artificial intelligence, network programming, operating systems, languages and algorithms. A modern computer game engine, such as Doom 3 [1] or Unreal Tournament 2004 [2] contains efficient, well-tested implementations of a wide range of powerful rendering and interaction techniques. These are generally focused on displaying realistic 3D "worlds" and supporting navigation within and interaction with elements in the visualisation. Given the power, flexibility and maturity of these game engines it makes sense to investigate ways of reusing such engines for other visualisation tasks, thus potentially saving large amounts of development time.

In this paper we focus on utilising computer game implementations for more general visualisation tasks such as information visualisation and scientific and biomedical visualisation. We present a framework for developing game engine based visualisation applications and we illustrate the necessary software architecture and its relationship to the data mapping process. An analysis of game engines is followed by a summary of our experiences obtained from several case studies.

2 Related Work

Computer game implementations have been successfully applied to visualisation related tasks such as architectural design critique [3], military simulations [4], and landscape planning [5]. Game engines have also been used for more abstract visualisation tasks. One innovative example is PSDoom [6], which is a utility for process visualisation and management, implemented as a modification of the Doom computer game. It provides the functionality of the Unix `ps` command via a 3D user interface. Running processes are represented as monsters (enemies), which can be shot and killed, thereby terminating the associated process. Monsters can fight back, and more important processes are represented by bigger monsters (which are more difficult to kill), thereby reducing the chance that they will be terminated. Interestingly, when many processes are running, and the 3D space becomes crowded with monsters, the monsters start attacking each other (a normal Doom behaviour). This provides a natural control mechanism for processes in a heavily loaded system - less important processes will be killed first, since the important processes are represented by stronger monsters.

3 Game Engines

3.1 Game Genres

Out of the computer game genres which use graphics (as opposed to text-based games) the main ones are First Person Shooter (FPS), Real Time Strategy (RTS) and Role Playing Game (RPG). Computer games can be classified further into single player, multi-player, or both.

In a FPS game, the player travels around in a three dimensional world, shooting enemies. In a RTS game, the player views a two dimensional map with many units (for example, of an army) on it. Players

control their own units and use them to attack and defeat their opponents. A RPG is similar, except the player controls only one unit, their "character", via which they explore a 2D or 3D world.

3.2 Game Engine Design

The typical high-level design of an engine is illustrated in figure 1. Most modern engines use a client-server architecture which supports multi player capabilities and distributes computing resources as follows [7]:

Server side: network scene management, server game play code, AI, static file I/O, world construction and layout, scripted content creation, database analysis and recovery, persistent storage.

Shared: low-level networking, collision detection/intersection, simulation/physics, entity layer, spatial partitioning and search, 3D animation (skeletal), script evaluator, geometry and animation exporters, game master tools.

Client side: network prediction/correction, client game play code, 3D animation (full), sound manager, streaming file I/O, physically-based audio/animation arrangement, 3D rendering/scene management, low-level 3D rendering.

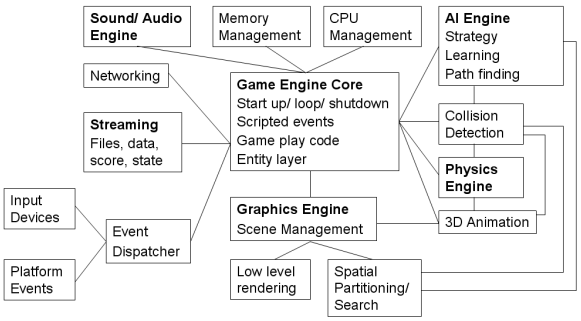

Figure 1: The high-level design of a game engine.

3.3 Game Architecture

Most modern computer games can be split into three parts: the game engine, the game logic and the game art. The game engine is the main executable program which runs on the computer. It provides an environment within which the game logic runs, as well as basic mathematics, graphics, audio, user input and network functions. The game logic may take the form of scripts, byte code for a virtual machine, or a library (a DLL for example). It controls the game play and uses the game engine to display the game art as appropriate. The game content (game art) incorporates the game play and media files such as pictures (textures in game parlance), maps (layouts of virtual worlds), models (3D representations of objects inhabiting the world, such as players and weapons) and sounds. Many developers incorporate their own proprietary formats for data, in particular 3D data, which has been optimised for use with the engine. The main benefit of utilising this type of game architecture is flexibility. It enables the developer to open-source the game code, letting users create modifications ("Mods") - anything from small weapon additions to an entire game - without the need to access the engine source code.

3.4 Available Game Engines

Game engines can be divided into two categories: open source and closed source. Open-source game engines are usually either written by amateurs or are older commercial engines. In the former category, some of the more popular engines are: OGRE [8], Crystal Space [9], Irrlicht [10], and The Nebula Device 2 [11]. In addition to these, there are the Doom, Doom 2, Quake and Quake 2 engines which have been open sourced by id Software [1] and use the OpenGL library for rendering. Since they are old commercial engines they include support for all gaming features such as physics, audio, network and GUI. None of the four amateur engines mentioned above have as much functionality as the id Software engines, however they may be combined with external libraries to provide the missing functions. The id Software engines suffer from being older, providing poorer rendering quality than the newer open source engines.

There are currently three main closed-source game engine families in the FPS genre, each from a different developer: Doom 3 and Quake 3 engines from id Software, Half Life and Half Life 2 engines from Valve Software [12] and Unreal Tournament (UT), Unreal Tournament 2004 (UT2004), and Unreal 3 engines by Epic Games [2]. Doom 3, Half Life 2 and Unreal 3 represent the latest generation from each developer, and are the best game engines available. All of these closed-source engines are fully-featured, and there exist one or more complete games based on each of these engines. This is in contrast to most of the amateur engines mentioned above, which usually need to be combined with other libraries and toolkits to create a playable game. Quake 3 deserves special mention as id Software has released the source code for this engine recently [13]. This means that, unlike the other engines here, extensive modifications to the game engine are possible ("Mods" for the other ones are restricted to altering the game logic and art).

4 Game Engines for Visualisation

Visualisation aims to represent complex data by graphical representations which convey information and understanding. Visualisation research is often divided into the fields of information, scientific, biomedical and software and algorithm visualisation.

4.1 The Visualisation Process

The visualisation process can be represented by a pipeline which performs *data encoding* and *data decoding* as shown in figure 2. The first stage of the data encoding step is the *data transformation* stage that converts information into a form more suitable for visualisation. This can involve creation of new quantities and subsets, data type changes, and modelling operations (e.g., model a directory structure as a tree). The subsequent *visualisation mapping* converts the transformed data into graphical representations which the *rendering* stage then displays. Some authors prefer to subdivide the mapping stage further into visual transformation (or data modelling) and visual mapping [14, 15]. However, in many applications these two stages are combined: the available models are fixed and the parameters of a model (shape, size, colour, texture) represent the encoded information. The data decoding step describes how visual information is perceived and processed and consists of visual perception and cognition.

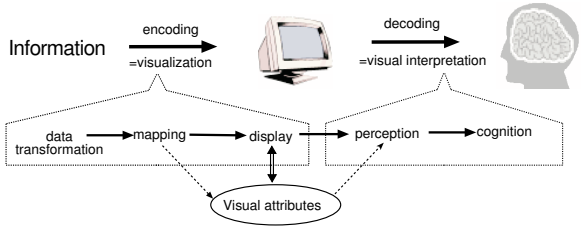

Figure 2: The visualisation process.

The encoding and decoding step of this schema are connected via *visual attributes* such as shape, position, and colour, and *textual attributes* such as text and symbols which themselves are represented by simple visual attributes. A visualisation is effective if the decoding can be performed efficiently and correctly. "Correctly" means that perceived data quantities and relationships between data reflect the actual data. "Efficiently" means that a maximum amount of information is perceived in a minimal time. The challenge is to achieve these goals while making the best use of the capabilities provided by the game engine without redesigning the engine source code.

4.2 A Framework for Game Engine Based Visualisation

There are two main ways in which a FPS game engine can be used for visualisation. One way is to modify an existing game which is implemented on top of the engine, and only add the features necessary for the visualisation, leaving the basic style of interaction with the 3D world intact. The other way is to write totally new code for the game logic, and only make use of the graphics, audio and networking functionality provided by the engine itself. While this approach is more flexible, it requires a lot more work on the part of the developer. In fact, this approach is similar to using a visualisation toolkit or engine, such as OpenSG [16]. Hence this approach may contradict the main purposes of using a game engine, such as code reuse, application of an intuitive interface, and use of a widely-used, growing, freely available software environment.

4.2.1 Software Architecture

It can be seen from figure 1 that game engines do not support the data transformation and mapping process. These tasks must be performed by a module sitting on top of the engine. Many data sets are extremely complex and in order to achieve interactivity the visualisation pipeline must be spread over the client-server architecture as illustrated in figure 3.

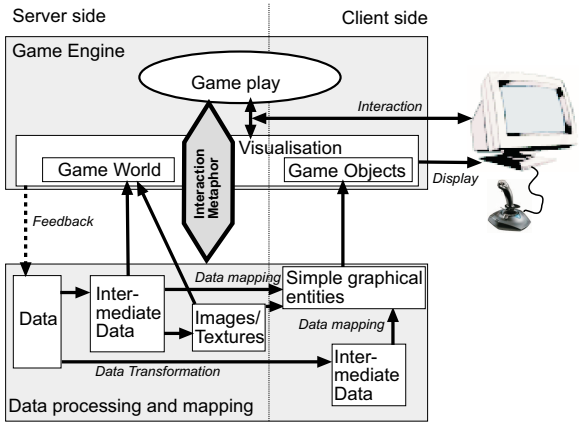

Figure 3: Software architecture for a game engine based visualisation tool.

If the data set is very large it should be processed on the server side and mapped to graphical entities which are then displayed on the client side of the game engine. Vice versa, if the original data set is relatively compact, but the graphical representation is very complex it will be more efficient to do the transformation and mapping stage on the client side. Note that the game world is usually stored on

the game server. Hence it might be useful to split visualisations into static components represented by the game world and dynamic components which are stored on the client side. Care must be taken to allow suitable interaction mechanism so that the user can go back from the visualisation to the corresponding data.

4.2.2 Data Mapping

In an FPS game, there are two primary types of elements: a static, or almost static, map (3D layout of rooms) and dynamic, interactive entities occupying positions in this map. The main challenge met when using a game engine for visualisation is that the set of available visual attributes, textual attributes and interaction techniques is limited. Hence creative approaches are required when transforming and mapping the raw data into graphical representations. The particular mapping chosen depends on the particular visualisation. For example, in a visualisation of a file hierarchy, the layout of directories can be represented by the layout of the rooms (that is, the map), while files are entities occupying positions within these rooms.

Limitations imposed by game engines must be taken into consideration when designing this mapping. One example is that in most current FPS games (specifically, Quake 3), the map can not be altered during a game session. This could be partly worked-around by altering maps while the players are in another map, creating the illusion of a dynamic world. This has the disadvantage that the game will pause while switching maps. Additionally, Quake 3 maps are limited in size. A solution is to split a large map into several smaller maps, but with the same problem of the game pausing between map changes.

Another limitation of FPS games is that they are designed for a relatively low number of entities; Quake 3 only allows a maximum of 1024 entities in a map. Thus, in some cases it may make more sense to represent parts of the visualisation as dynamically generated textures (e.g., a diagram of a graph structure) rather than as separate entities. An alternative solution is to use multiple maps with the disadvantages discussed above.

Yet another peculiarity of game engines is that they are designed to only support one style of interaction defined by the game logic. For example, in Quake 3 each entity usually has a fixed appearance, and a fixed behaviour throughout a game session. (It is actually possible to alter these during a game session by programming the game logic, if desired.) The problem is that the engine does not provide any "multiple view types" support. So, if the visualisation to be implemented relies on multiple view types, one must code a framework on top of the engine which keeps track of what view is being currently used, and tell the game logic which representations and behaviours to use for which entity.

4.3 Analysis of Game Engines

In order to create an easy-to-use, collaborative visualisation tool the utilised game engine must be freely available, multi-user capable, stable, well tested and feature rich. Furthermore we want to avoid modifying the source code, hence the game engine needs a powerful, flexible scripting language and there must be a well-tested, open-source, implementation of a game for it available.

We have evaluated multiple game engines [17, 18] and found that the Quake 3 engine, with the game implementation Quake 3 Arena [19, 20], meets these requirements best. The Quake 3 Arena source code is available, under a limited licence (which does appear to permit modifying the source code and distributing the modified game virtual machine byte code).

Several open source game engines, such as OGRE, Crystal Space and Irrlicht, were investigated, however most of them lacked crucial features (such as networking support), or had no well-tested game implemented using them. Quake 3 uses the standard FPS game control system: mouse and keyboard. Moving the mouse around changes the direction the player looks in. The mouse buttons are typically used for walking forwards and for shooting. Various keys on the keyboard are used for crouching, jumping, moving backwards, strafing and switching weapons. The keys can be remapped to different in-game functions via a process known as key binding. Quake 3 is by design a network-oriented (LAN or Internet) game, using the client-server model of communications. Each computer running Quake 3 runs an instance of quake3.exe, the game engine. This executable is capable of running byte code for three virtual machines: *game*, *cgame* and UI. These are referred to as QVMs, for Quake Virtual Machine. The *game* qvm is the server part of the game. It is responsible for maintaining the state of the game world, such as positions of all entities, and sending messages to the clients. It also has the final say on issues such as whether a certain bullet hit a certain player or not. *game* does not do any rendering; it only communicates with clients. *cgame* is the client QVM - there is one running on each computer connected to a particular game. The client is responsible for

rendering the map and entities, according to data sent by the server. Finally, the UI qvm is responsible for displaying the in-game menus. Within the QVM environment there are several available system calls or traps. These are functions that can be called within the code of the QVM, to pass control into the main quake3.exe executable. This is how tasks such as drawing on the screen, file and network access are carried out. Internally, the main message passing mechanism (or rather, the most easily accessible and modifiable one) is based on passing variable-length, null-terminated strings.

5 Results

We found that Quake 3 is a good choice for implementing game engine based visualisation applications and we successfully implemented a software visualisation tool [17] and a biomedical visualisation tool [18] which are depicted in figure 4.

Figure 4: A software comprehension tool (bottom) and a biomedical visualisation tool (top) implemented with the Quake 3 engine.

Learning to modify the game engine took time, and was mostly done through trial and error. There is no documentation of the code provided by id Software and it was often necessary to walk through the existing code by hand. The game code which runs on the VMs is written in C. This has the usual side-effect of having the implementation of a particular entity split across many different files. This is not necessarily a bad thing in itself, but sometimes this leads to subtle bugs where some other part of the code in another file unexpectedly alters the state of an entity.

An extremely useful feature in Quake 3 is the extensive *shader* system. The shaders in Quake 3 allow the creation and application of dynamic materials in-game. There are numerous effects, including 2D translation, alpha blending, and environment mapping, all of which are configurable by editing a `.shader` script file. This comes in handy when implementing graphical glyphs in the visualisation. An added bonus is that several tools exist to visually create these shader file.

Another benefit of the Quake 3 engine is the *console*. This is an area in-game which allows the user to input commands, or adjust variables. Commands can be useful functions such as `screenshot`, allowing the user to capture the current screen, whereas variables can have instant feedback in the engine - for example, setting `r_showtris` to 1 allows us to see every triangle drawn on screen.

The most significant problem for visualisation purposes is that the MD3 model data format used in Quake 3 seems to limit the maximum number of vertices of a surface mesh to 4096. This made it necessary to split complex biomedical models, such as the pelvis in figure 4 (top), into separate meshes which were then exported as a single MD3 file.

Another problem is that the network protocol is fixed. New character string messages can be added easily, but these need to be assembled at one end, and parsed at the other. In addition, these messages are not associated with any particular entity, which makes it difficult to communicate data specific information from the server to the clients. Fortunately, a field in the existing packet structure was found which was rarely used, and so was reused to transmit the *itemid*. While this worked for this particular tool, the available space is very limited, and may not be sufficient for other visualisation tools.

6 Conclusion

It is possible to use a game engine as the basis for a visualisation tool, and thereby save a lot of implementation time by reusing the functionality already implemented in the game engine. The biggest benefit is obtained by creating a mod rather than rewriting the source code. Modern game engines offer a high speed, rendering quality and interactivity and multi-user support which

is difficult to obtain using existing visualisation tools.

In order to simplify the implementation process, it is important to find a suitable visualisation and interaction metaphor which often requires considerable creativity. Due to the limitation of most engines the visualisation might have to be split into a static 3D world and dynamic entities that the users can interact with (e.g., obtain feedback from). There should typically be much less than 500 of these entities, but this limit depends on the game engine chosen, the amount of modifications needed, and the performance and hardware requirements. There typically is also a limitation on the maximum allowed map (3D world) size and the size of surface meshes, although this can be usually worked around by stitching together several smaller maps and meshes, respectively.

Developing visualisation applications using game engines requires considerable more work than when using a visualisation library such as VTK [21]. We hope that this research will enable the reader to design visualisation tools which sit on top of a game engine and combine the advantages of standalone visualisation applications with that of game engines.

References

[1] "id Software." URL: `www.idsoftware.com`.

[2] "Epic Games." URL: `www.epicgames.com`.

[3] J. Moloney, R. Amor, J. Furness, and B. Moores, "Design critique inside a multiplayer game engine," in *Proceedings of the CIB W78 Conference on IT in Construction*, pp. 255–262, 2003. Waiheke Island, New Zealand, 23-25 April.

[4] J. Manojlovich, P. Prasithsangaree, S. Hughes, J. Chen, and M. Lewis, "UTSAF: A multi-agent-based framework for supporting military-based distributed interactive simulations in 3d virtual environments," in *Proceedings of the Winter Simulation Conference*, pp. 960–968, 2003. New Orleans, 7-10 December.

[5] A. Herwig and P. Paar, *Trends in GIS and Virtualization in Environmental Planning and Design*, ch. Game Engines: Tools for Landscape Visualization and Planning?, pp. 161–172. Wichmann Verlag, Heidelberg, 2002.

[6] D. Chao, "Doom as an interface for process management," in *Proceedings of SIGCHI'01*, pp. 152–157, 2001. Seattle, WA, 31 March-1 April.

[7] J. Blow, "Game development: Harder than you think," *ACM Queue*, vol. 1, no. 10, pp. 58–65, 2004.

[8] "OGRE Object-oriented Graphics Rendering Engine." URL: `www.ogre3d.org`.

[9] "Crystal Space 3D." URL: `crystal.sourceforge.net`.

[10] "Irrlicht Engine - A free open source 3d engine." URL: `irrlicht.sourceforge.net`.

[11] "The Nebula Device 2." URL: `nebuladevice.cubik.org`.

[12] "Valve Corporation." URL: `www.valvesoftware.com`.

[13] "Quake 3 1.32 Source Code," Aug. 2005. URL: `http://www.fileshack.com/file.x?fid=7547`.

[14] E. H. Chi, "A taxonomy of visualization techniques using the data state reference model," in *Proceedings of the Symposium on Information Visualization (InfoVis '00)*, pp. 69–75, IEEE Press, Oct. 2000.

[15] C. Ware, E. H. Chi, and R. Gossweiler, "Visual perception for data visualization (tutorial for CHI 2000)," in *Tutorial for the Human Factor in Computing Systems Conference (CHI 2000)*, (Amsterdam, Netherlands), Apr. 2000. URL: `http://www-users.cs.umn.edu/~echi/tutorial/perception2000/`.

[16] "OpenGS Homepage." URL: `www.opensg.org`.

[17] B. Kot, "Information visualisation utilising 3d computer game engines," FoS Scholarship report, Dept. of Computer Science, University of Auckland, Feb. 2005. URL: `www.cs.auckland.ac.nz/~burkhard/Reports/SS2004_BlazejKot.pdf`.

[18] A. Gits, "Using game engines for visualising biomedical data sets," 780 Graduate Project report, Dept. of Computer Science, University of Auckland, June 2005. URL: `www.cs.auckland.ac.nz/~burkhard/Reports/2005_S1_AndrewGits.pdf`.

[19] "Code3Arena." URL: `www.planetquake.com/code3arena/`.

[20] "Quake III: Arena, baseq3 mod commentary." URL: `www.icculus.org/~phaethon/q3mc/q3mc.html`.

[21] "VTK Home Page." URL: `public.kitware.com/VTK`.