MAGE: A FLEXIBLE, PLUGIN-BASED APPROACH TO GAME ENGINE ARCHITECTURE

by

AMIT MATHEW

(Under the Direction of Jeffrey Smith)

ABSTRACT

MAGE is an open-source game engine that uses third-party middleware components to provide functionality for its game subsystems. This approach allows developers to choose best-of-breed components that meet their requirements, instead of being restricted by the components an individual game engine provides. These components can be superior to those that are part of game engines that are built from the ground up, since each component is specialized for one specific function. This approach is implemented by providing an extra layer of abstraction between the game engine and the middleware, which is shown to add only a minor performance penalty.

INDEX WORDS: MAGE, game engine, game development, game engine architecture, virtual reality, game middleware

MAGE: A FLEXIBLE, PLUGIN-BASED APPROACH TO GAME ENGINE ARCHITECTURE

by

AMIT MATHEW

B.S., University of North Carolina at Chapel Hill, 2002

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2008

MAGE: A FLEXIBLE, PLUGIN-BASED APPROACH TO GAME ENGINE ARCHITECTURE

by

AMIT MATHEW

DEDICATION

To Veena, Appa, Amma, Arun, Anjali, Daddy, Mommy, Chach, and Viji. I am fortunate to have a list so long!

ACKNOWLEDGEMENTS

This thesis was a long time coming, but hopefully it was worth the wait. I give a huge thanks to Dr. Smith for agreeing to be my major professor and for promoting game development at UGA and to Dr. Deligiannidis for his guidance throughout my graduate career. I'd also like to thank Dr. Everett and Dr. Miller for being on my committee.

I would have never made it to this point without the amazing work of the OGRE team and the ever helpful OGRE community. They are the reason I get paid for doing stuff that I would do for free and the reason I have something interesting to write about for a thesis.

And thanks to my family for putting up with all this. See, all those years of playing games paid off!

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A game engine is a software component that controls the basic subsystems of a video or computer game. Common game systems include graphics, audio, physics, collision detection, artificial intelligence, networking, and input. Game engines are often, but not always, separated from the game logic so that different games can be written with the same engine. A game engine is combined with game logic and artistic content (such as 3D models, textures, music, sound effects, and scripts) to form a complete game.

Game engines are sometimes written specifically for a certain game, and other times they are designed for a particular type of game genre. The Unreal Engine [42], for example, is primarily designed for first-person shooters (although it has been used in other types of games). Others are written to be more general-purpose, such as Gamebryo [14].

This work describes the background, software architecture, and performance characteristics of MAGE, a general-purpose game engine. MAGE takes a different approach than many other game engines by using an architecture that allows external libraries to load at run-time and to supply the functionality of all the game systems. This is known as a plugin-based architecture. MAGE ties all the systems together into one cohesive package. This approach allows developers to choose existing third-party components known as middleware, which fit their needs the best, while still using the MAGE interface.

MAGE uses an architecture that enables the use of highly specialized middleware libraries. The reasoning behind this is that the best libraries focus on doing one thing and doing that one thing well. The performance evaluation section of this work tries to determine the validity of one important part of that assumption by comparing the performance of MAGE to other game engines that are similar in scope.

CHAPTER 2

RELATED WORK

The design and architecture of game engines are well understood in commercial circles, but the amount of research in academia on this subject is limited. Although there is a wealth of publications in the areas related to game development such as computer graphics, audio processing, networking, artificial intelligence, and rigid-body dynamics, there has been little work done with games themselves and how they interact with these other fields. Part of the reason for this is that game development hasn't been taken seriously as an academic topic until recent years. As researchers have realized that games can be used for purposes other than entertainment and that game development poses interesting and unique theoretical problems, the amount of interest in the field has increased.

A good overview of game engine development is presented in the paper describing the NetImmerse engine [23]. This article gives a good overview of the subject and the challenges facing anyone attempting to design a game engine. NetImmerse is designed to be a general-purpose game engine, one that is not tied to a particular genre. NetImmerse provides a layer of software that hides the underlying implementation details of OpenGL [25] and DirectX [9]. This is known as an abstraction layer. NetImmerse uses a scene graph, a hierarchical data structure used for efficiently culling graphical objects that are not in view, for scene organization.

A work that deals with the general software architecture of game engines is Jeff Plummer's master's thesis [21]. In it, he describes the concept of a "system of systems", where off-the-shelf components are loosely coupled to form a complete game engine. This is a similar approach that MAGE takes. Plummer explains the architecture in great detail via UML diagrams and through a prototype system he created.

The design and architecture of a more complete game is described in Hadwiger's Master's thesis [26]. In the thesis, he describes the Parsec game engine, which was created for a multiplayer space combat game. His engine differs from MAGE because it is very specialized for the particular type of game being made, whereas MAGE is more a general-purpose engine. Hadwiger's results are compelling since his game was fully playable and had an extensive user base.

In addition, there are several books on game development. Most of these book focus on developing the individual components present in a game engine, so they usually have quick overviews of computer graphics, physics, etc. A book that focuses more on the game engine itself is David Eberly's work [8]. It describes the design of his Wild Magic software and provides a fascinating look at the development of a high-performance game engine. Although it contains information specific to fields like graphics, it also contains relevant information about game engine architecture in general.

Panda3D [28] is the software system that probably most closely resembles MAGE's feature set. Initially developed for a Disney massively multiplayer online game, it is now open source software that is co-developed by Carnegie Mellon University's Entertainment Technology Center and Disney. In addition to the Disney game, it has been used for class projects at Carnegie Mellon. It is under active development and focuses on ease of use and rapid development.

VR Juggler [2] is one of the most prominent Virtual Reality (VR) application development frameworks. It was developed at Iowa State University's Virtual Reality Applications Center. VR Juggler runs on a number of different platforms and supports a wide variety of VR devices. Like MAGE, it offers a variety of modular components that implement various pieces of functionality. Although not a game development platform, VR-Juggler does allow the development of virtual environments (VEs) like MAGE-VR [4].

CHAPTER 3

GAME ENGINE COMPONENTS

Much of a game engine's functionality is defined by its components. Although the types of components in a game engine can vary, there are several components that are common to most modern game engines. The following sections describe the various components and how they affect MAGE. Note that currently the game artificial intelligence, networking, scripting, and graphical user interface systems are not implemented in the present version of MAGE. They will be added in future releases (see Ch. 7 for details).

3.1 Core

The game engine core is responsible for coordinating the various systems and providing functionality like logging, math routines, timers, and other utility functions used throughout the engine. The core uses the concept of notifying objects of events, known as message passing, to keep different systems synchronized. For example, when a game object's transform (its position, rotation, and scale) changes, its graphical, audio, and physical components need to be notified. The core also is responsible for correctly initializing and de-initializing the different systems in the proper order. In MAGE, the core loads systems as dynamically linked libraries on demand, which can be controlled by a configuration file or programmatically.

3.2 Resource Manager

A resource manager is responsible for managing the external assets used by a game. These assets can be artistic assets like 3D models, textures, or music files, or they can be scripts, which define certain behaviors during gameplay. Because game resources are often large and take up significant memory, they are usually loaded on demand and unloaded again when they are no longer needed. Some games load all the resources for a level at the beginning of a level, using a loading screen and making the user wait until the loading is completed. Other games load everything on the fly, so that the user doesn't have to sit through a loading screen. This is usually accomplished by having a background thread that is constantly loading and unloading resources. This approach provides a more seamless experience for the user, but it entails more technical complexity and causes poorer run-time performance because of the overhead of the background thread. Without proper resource management, games can suffer from long loading times and poor performance.

Resource managers often have to manage data from three primary sources. The simplest case is when the resource is on a hard disk drive. If the resource is small enough, it can be loaded entirely into memory. If it is not small enough, it must be streamed from the disk. This is often case for music files, which can be several megabytes in size. Another source is external media, which today is usually CD-ROMs or DVDs. Since these data sources are often much slower than hard disk drives, resources are usually copied to the hard disk drive when the game starts or, in the case of game consoles that don't have hard disk drives, they are streamed as the game is running. The third source of resource data is the network. For some games, resources may be stored on a server and they must be downloaded before they are used. Since remote servers are much slower than the previous two sources, the resources from servers are usually downloaded in advance.

Additionally, resources can come in different forms. Resources are often compressed when they are stored on disk and uncompressed as they are loaded into memory. Since a modern game's resource set are often in the gigabyte range, it becomes necessary to compress them. Game resources are also often encrypted so as to protect the assets from people who may try to hack the game or steal the content. This is very important in multiplayer games where cheating can ruin the gaming experience for many people.

MAGE currently has basic resource management support for graphics and audio files. It can load both compressed and uncompressed resource from the local disk. Since all loading and unloading of resources in MAGE is explicit, it does not currently have an intelligent system that automatically unloads resources when memory is scarce.

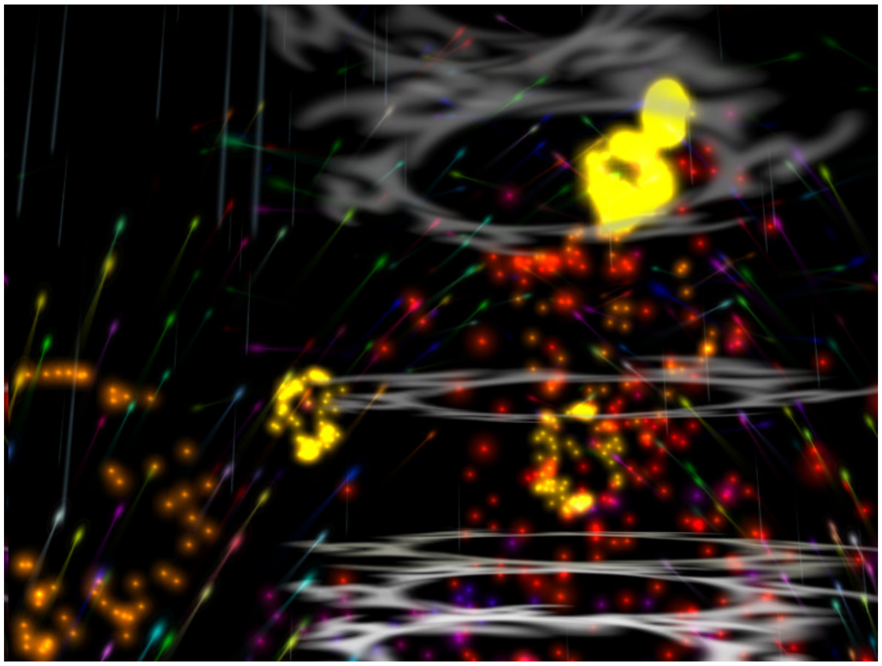


Figure 1: A particle effects demo of MAGE using the OGRE graphics system.

3.3 Graphics System

The graphics system is responsible for managing the visual output of the game. Graphics can be three-dimensional, two-dimensional, or text. Most modern games are three-dimensional, although some games use a fixed-position camera to simulate two-dimensionality. Graphics systems can usually display a wide variety of object types, such as 3D models, lights, shadows, textures, and particle systems. Figure 1 shows an example of some of these effects. Graphics engines are often the most complex aspect of a game engine, so many game engines are designed around the needs of the graphics engine.

MAGE only supports 3D graphics engines, and furthermore stipulates that the engine must be scene graph-based. Scene graphs are discussed in detail in section 4.8.

3.4 Audio System

Audio systems manage the aural outputs of a game. These can include sounds, character dialog, and background music. Smaller sound files can usually be loaded into and directly played from memory, but longer audio clips, such as character speech and songs, often need to be streamed from the disk. Modern audio systems can simulate 3D sounds, where sounds can seem to come from different directions in relation to the listener. Advanced audio systems can also simulate the Doppler effect, which is the phenomenon where the pitch of a sound drops sharply after the source passes the listener.

MAGE supports both streamed and unstreamed audio. It supports 3D audio and the Doppler effect for unstreamed sounds. MAGE can also play a wide variety of audio file formats.

3.5 Input System

The input system is the mechanism through which the player controls the game. Traditional computer games supported keyboards and mice, but modern games can additionally support console-style controllers, joysticks, and motion-sensing controls. Input systems usually support customization, so that the player can select the effect of each input signal. MAGE currently supports just basic keyboard and mouse inputs.

3.6 Collision Detection System

The collision detection system reports collisions between objects in the game world back to the game engine. This information can be used in several ways. One common use is keeping game objects on the ground and keeping them from going through world objects. The collision detection system can also be used for game artificial intelligence, so that an AI-controlled character can simulate vision by casting a ray in the direction she is facing. If the ray intersects an object, then the AI character can "see" that object.

The collision detection system is often integrated with the physics system, which is the approach that MAGE uses. Collision detection can be considered to be the first phase of physics simulation, because physics is usually simulated after it is determined that objects have collided.

Figure 2: MAGE running a physics simulation of tumbling boxes using the Newton physics library.

3.7 Physics System

The physics system simulates the effects of physics on game objects. The most basic form involves simulating the behavior of object collision, but advanced physics systems can simulate joints, object breakage, and springs. There are two general types of physics simulations: rigid body and soft body dynamics. Rigid body dynamics can only use objects that do not deform, but results in less computationally expensive physics calculations. Soft body physics can simulate object deformation, but their use in current games is infrequent because of the computational costs.

MAGE currently supports basic physics simulations of game objects' responses to collision. MAGE also allows developers to tune several parameters that give fine-grained control of how an object behaves after a collision. These parameters include mass, gravity, softness, and bounciness. Figure 2 shows MAGE using the Newton physics system in action.

3.8 Game Artificial Intelligence System

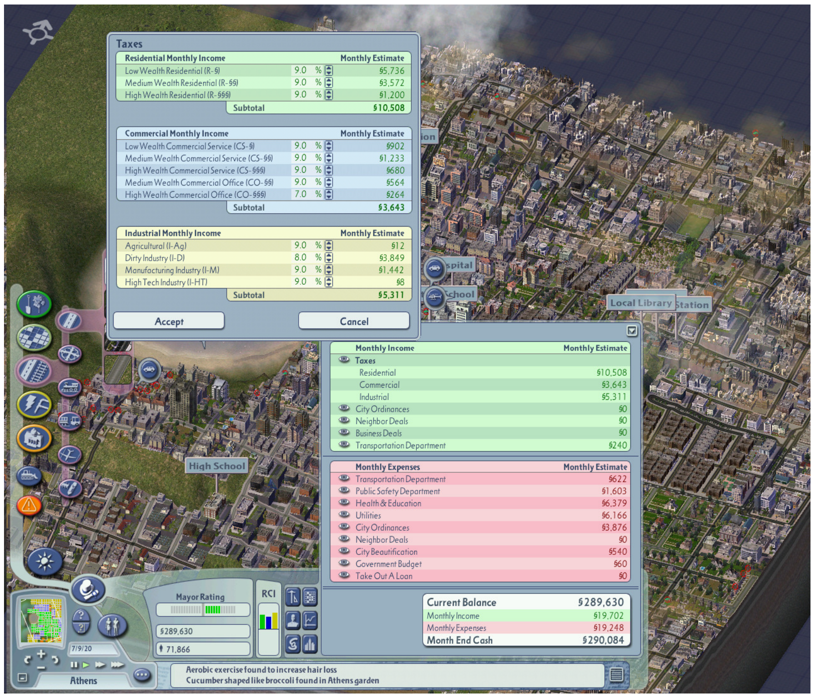
Game artificial intelligence (AI) is a system that gives the characters in the game the appearance of intelligent behavior. Game AI is different than traditional AI because characters in a game don't necessarily need to be intelligent. Instead, they must merely give the impression to the user that they are behaving intelligently. This can be accomplished by using shortcuts based on the system's knowledge of the game world. For example, a game designer may mark certain game engines as objects that provide good cover, so that when an AI-controlled character comes under fire, he might hide behind a designated cover object. This appears to the user that the character was intelligent enough to know the object provided cover, but in reality this was pre-designated.

3.9 Networking System

A networking system allows a player to play against others using network communication. Depending on the requirements of the game, the network system may use a client/server architecture, a peer-to-peer architecture, or some combination of both. Games also may support anywhere from two to several thousand players in a game world, depending on the game type. A networking system must remain responsive with a congested network, or the quality of the user experience will degrade dramatically. Furthermore, a network system usually provides some level of cheat prevention so that one player cannot ruin the experience of others.

3.10 Scripting System

Many games provide some sort of scripting system so that designers can write game logic by designers in a high-level (and usually interpreted) programming language. Since game logic is usually not a performance bottleneck, the use of a slower language does not usually impact the overall game performance significantly. The benefit is that designers can rapidly iterate through different designs without worrying about a long edit-compile-debug process.



Taxes

Residential Monthly Income		Monthly Estimate
Low Wealth Residential (R-§)	9.0 %	§5,736
Medium Wealth Residential (R-§§)	9.0 %	§3,572
High Wealth Residential (R-§§§)	9.0 %	§1,200
Subtotal		§10,508

Commercial Monthly Income		Monthly Estimate
Low Wealth Commercial Service (CS-§)	9.0 %	§902
Medium Wealth Commercial Service (CS-§§)	9.0 %	§1,233
High Wealth Commercial Service (CS-§§§)	9.0 %	§680
Medium Wealth Commercial Office (CO-§§)	9.0 %	§564
High Wealth Commercial Office (CO-§§§)	7.0 %	§264
Subtotal		§3,643

Industrial Monthly Income		Monthly Estimate
Agricultural (I-Ag)	9.0 %	§12
Dirty Industry (I-D)	8.0 %	§3,849
Manufacturing Industry (I-M)	9.0 %	§1,442
High Tech Industry (I-HT)	9.0 %	§8
Subtotal		§5,311

Monthly Income	Monthly Estimate
Taxes	
Residential	§10,508
Commercial	§3,643
Industrial	§5,311
City Ordinances	§0
Neighbor Deals	§0
Business Deals	§0
Transportation Department	§240

Monthly Expenses	Monthly Estimate
Transportation Department	§622
Public Safety Department	§1,603
Health & Education	§6,379
Utilities	§6,166
City Ordinances	§3,876
Neighbor Deals	§0
City Beautification	§540
Government Budget	§60
Take Out A Loan	§0

Current Balance	§289,630
Monthly Income	§19,702
Monthly Expenses	§19,248
Month End Cash	§290,084

Aerobic exercise found to increase hair loss
Cucumber shaped like broccoli found in Athens garden

Figure 3: A fairly complex GUI from Maxis's SimCity 4.

3.11 Graphical User Interface System

Graphical user interfaces (GUIs) in games usually mimic traditional 2D style GUIs used in operating systems, but are often rendered in a 3D environment. GUIs in games range from the very basic, for games that use GUIs for simple menus, to the advanced, such as in detailed simulation games where the user is presented with a multitude of options. Figure 3 shows an example of one such complex game GUI, with hierarchical windows and many buttons. Like the scripting system, a GUI system is absent from MAGE and may be added as part as a higher-level game library in the future.

CHAPTER 4

SOFTWARE ARCHITECTURE

4.1 History of Game Engine Architecture

The video game industry began in the 1970's with big releases such as Pong [37] in 1972 and Asteroids [5] in 1979. Some games during this time, like Pong, did not have any software because the hardware on which they ran lacked microprocessors and the games were built directly into the electronic components of the hardware. For games that did have microprocessors, their software architecture consisted of monolithic programs written in assembly language. This type of architecture was forced by the constraints of the software and hardware of the time. The machine code generated by the mainstream high-level languages like Fortran and Cobol was not efficient enough for the limited hardware of early arcade machines. At the time, CPU clock speeds were low and RAM capacity was small. In such a constrained environment, it was necessary for a programmer to hand-tune each game for the particular hardware the game ran on. Also, since the early games were on arcade machines, each new game usually ran on different hardware and therefore each game had to be rewritten to run on the new hardware. Finally, the games of the time were fairly simple and were developed by one or two programmers. Since games were simple, there was not a need to put much thought into software architecture.

Although the first traces of game engines were developed in the 1980's with LucasArts' SCUMM [39] and Infocom's Z-machine (a text game engine) [18], the games of that decade were predominantly written either in pure assembly or some mix of C and assembly; C was the first programming language to gain widespread acceptance with game developers because of its focus on high performance. Software reuse was still typically minimal, perhaps due to the lack of hardware abstraction and standards. Another possible contributing factor to the lack of reuse is that game engines typically are written for a particular genre and video games were too young to have such delineated genres.

The birth of the concept of the modern game engine arguably occurred with the release of id Software's *Doom* [10] in 1993. *Doom* popularized the idea of licensing game engines, where companies would pay another company to use their game engine to create their own game. This approach saved developers time and money because they could focus on creating their game instead of developing the underlying components. *Doom*'s model of development continues today, with game companies releasing games using their proprietary game engines, which they then license to other companies.

In the intervening time, there have been significant changes to the field of game development. The programming language of choice has transitioned from C to C++, evidenced by the fact that *Doom 3*, released eleven years after the original, was written in C++ [11]. The transition to C++ was necessary for the industry because as games grew more complex, with source code lines numbering in the hundreds of thousands, it became more difficult to manage monolithic C programs without the benefits of modularization, inheritance, data encapsulation, and polymorphism that object-oriented programming offers.

Another important trend that has surfaced in recent years is the idea of middleware components. Instead of licensing entire game engines, companies now can license individual game engine components that they combine together to create a custom game engine. The benefits of this approach are flexibility and specialization. Licensing entire game engines usually dictates the type of game that can be developed, whereas individual components can be used to create a greater variety of games. Also, middleware components tend to be more powerful because middleware companies usually specialize in developing just one component, which they can focus their energies on. Products like Havok [17] (for physics) and fmod [13] (for audio) are examples of popular middleware components. Middleware has become so specialized that there exists components solely for rendering trees [40].

But middleware is not restricted to individual components. Products such as Renderware [38] and Gamebryo [14] are general-purpose game engines that, theoretically, can be used for any type of game. These products may not have the same power as genre-specific game engines, but they offer developers much more flexibility.

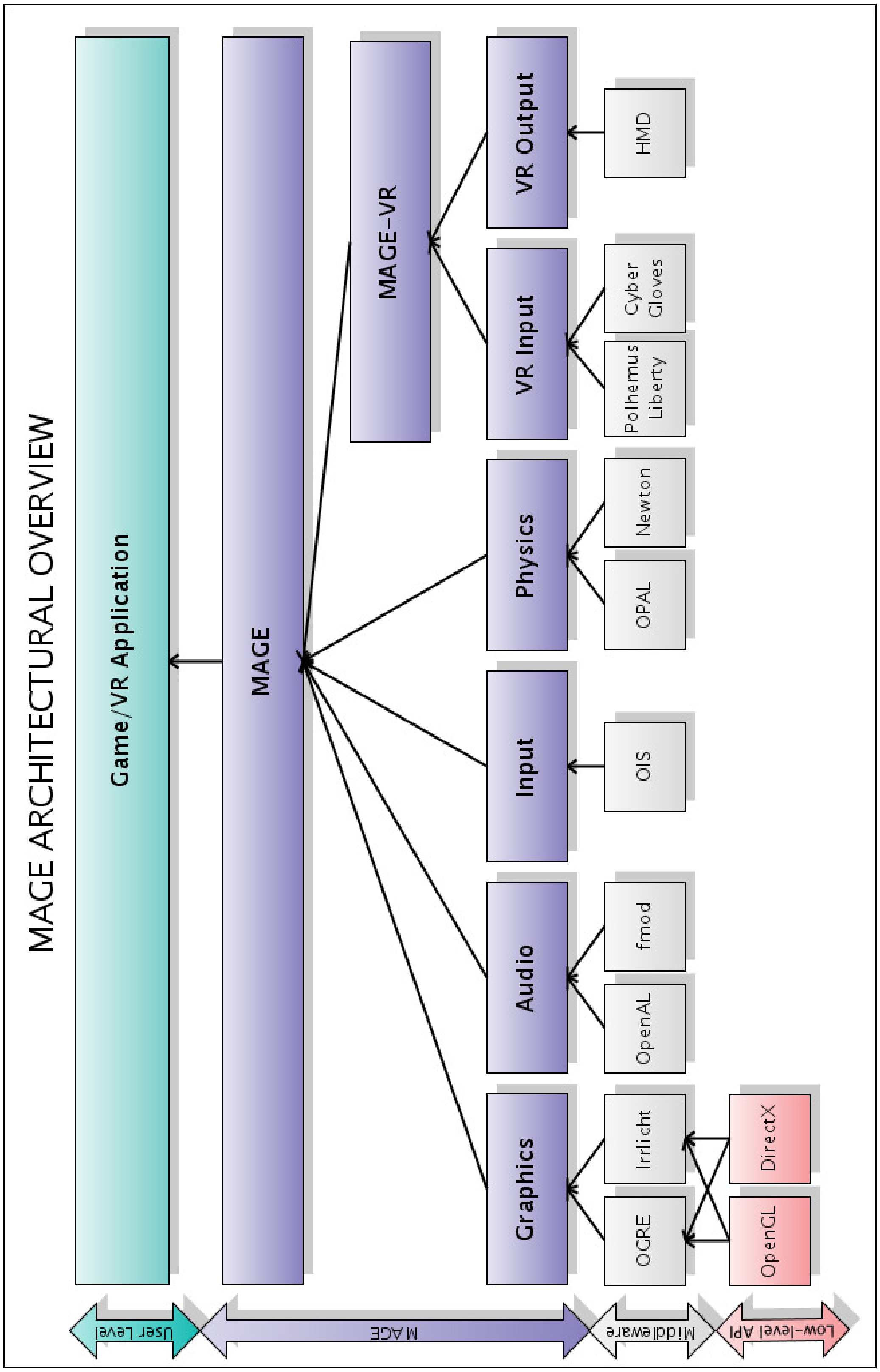


Figure 4: MAGE Architectural Overview.

4.2 MAGE's Plugin-Based Architecture

MAGE follows the example of commercial general-purpose game engines by providing a system that can be used to develop many common types of game. However, MAGE takes the concept of middleware one step further by using middleware components to provide the actual game engine functionality. Furthermore, MAGE does not limit which middleware components can be used, instead allowing the possibility for many components to be integrated, as long as an interface that allows one component to communicate with another component, known as a wrapper, can be written to plug the middleware into the MAGE system. Thus, MAGE introduces a new type of game engine framework, one that does not provide any implementation on its own, but allows the user to tie various pieces of middleware into one cohesive package. In a sense, MAGE is not a game engine in itself, but a framework for building customized game engines, and is similar in concept to Plummer's idea of a "system of systems" [21]. An overview of MAGE's architecture is shown in Figure 4.

The concept of a game engine framework supported by multiple middleware components is manifested in MAGE in the form of systems and plugins. Systems are the parts of the game engine that provide a distinct type of functionality. In MAGE, some of the supported system types are graphics, audio, physics, and input. Other possible types of systems are graphical user interfaces, artificial intelligence, game scripting, and content management. These systems will be detailed in the future work section. Each system in MAGE is given a specific interface and purpose, so the coupling between each system in minimized. Each system's implementation is provided by middleware components in the form of plugins.

In MAGE, plugins are dynamically linked libraries that are loaded at run-time to provide the implementation for specific systems. The power of MAGE's plugin system is that the user can choose which plugin to use for each system. Additionally, the user can choose not to use plugins for certain systems, if the functionality for those systems that are not required by the game. Since the plugins are loaded at run-time, the plugins can be switched on the fly without recompiling the application.

MAGE's plugin-based architecture offers developers increased flexibility. Before development of the game has begun, the development team can choose which libraries to use with MAGE. If a developer wants to use a library that isn't currently supported by MAGE, the developer can wrap the library in a plugin format that conforms to the MAGE interface to allow the library to be used within MAGE. This flexibility is important, because middleware components for the same system have different strengths. For example the Irrlicht [19] engine is extremely fast, but it doesn't offer the cutting edge graphical features of OGRE [31].

Another aspect of the flexibility of MAGE's plugin-based architecture is that development teams can switch components mid-development. If the developer wants to switch audio libraries during a project, this can be done by changing one line in a script file and, if the audio library is not currently supported, by writing a wrapper to for the new audio library. The same thing can be done with any of MAGE's systems. Since new middleware components are released frequently, the ability of MAGE to adapt to new third-party libraries is important.

4.3 Tradeoffs for MAGE's Flexibility

Although switching middleware components in MAGE is literally a matter of changing one line in a text file that specifies different parameters, known as a configuration file, some effort must be spent to use the new piece of middleware correctly. The reason for this is because the behavior for each component is different. The volume of a sound will vary from one audio library to the next. Physics engines require different tuning parameters to get realistic and consistent behavior. Switching graphics engines, in particular, requires extra effort because usually all the artistic content must be modified for the new engine. But in general, MAGE's design assumes the user may want to switch libraries mid-development.

Another significant design issue that faces the development of the MAGE is creating a common interface to the heterogeneous third-party middleware libraries without sacrificing features of the underlying libraries. The gap in the feature set between various libraries usually varies depending on the complexity of the implemented system. Graphics libraries, for example, tend to have the most variance in their features, whereas input libraries have the least.

MAGE's solution to this problem is to expose as many features as possible from the underlying middleware components. If a supported library does not have one or more features of one of the other supported libraries, the plugin for that library will throw an exception if the user attempts to use that piece of functionality. A concrete example of this behavior is with the graphics system's animation blending, which is a method of smoothly combining animations. OGRE supports animation blending, while the version of Irrlicht used in MAGE does not. Rather than not exposing this feature to the user, MAGE allows the user to attempt to perform animation blending, but if the Irrlicht plugin is being used, the plugin will throw a special MAGE_EXCEPT_UNSUPPORTED exception.

Another way that MAGE handles disparate functionalities is by providing data files to game objects that have many parameters. Data files, in this case, are files that specify the properties of an object. Data files are used for the material systems in OGRE and Irrlicht, because they have many parameters, most of which don't overlap. Rather than creating a different set of functions that would only be compatible with one library but not the other, MAGE adds a material data file syntax and parser to Irrlicht (but not to OGRE, since OGRE already supports material data files). The material data file specifies attributes like the textures used, ambient lighting, and the type of alpha blending to use. MAGE also supports specifying meshes and material systems with data files for Irrlicht.

4.4 Middleware Supported by MAGE

The key to MAGE's success depends on the quality of the middleware components it uses for its plugins. MAGE makes use of OGRE [31] and Irrlicht [19] for graphics, OIS [32], OGRE, and Irrlicht for input, OpenAL [34] and fmod [13] for audio, OPAL [33] (which in turn uses ODE [30]) and Newton [29] for physics, the Polhemus Liberty [36] driver for the motion tracker, and the CyberGlove [7] driver for the virtual glove. It should be noted that MAGE offers two different plugin implementations for most of its systems. This is to ensure that MAGE's interface for these systems are generalized enough to encompass more than one system. It is also worth noting that many of the underlying libraries are available at no cost, allowing MAGE users to develop games at little to no cost.

4.5 Virtual Reality Extensions to MAGE

MAGE-VR is an extension to MAGE that adds support for technologies typically associated with Virtual Reality (VR). MAGE treats VR devices as specialized input and output devices. This abstraction allows MAGE to support VR capabilities without many modifications to the core library.

The extensions to MAGE that comprise MAGE-VR are simply a few additional systems and plugins that are treated just like any other system within MAGE. In MAGE-VR, the additional systems are input systems for virtual reality gloves and 3D trackers. Virtual reality gloves are gloves equipped with sensors that determine the position and rotation of the wearer's fingers, palm, and wrist. A 3D tracker is a sensor placed on a user to determine their position and rotation in space relative to the base station. The plugins that implement these systems support the Immersion CyberGlove and the Polhemus Liberty motion tracker. Once these systems are in place, they are included seamlessly with MAGE's existing systems. For example, in the sample VR application included with the MAGE distribution, the user can use the CyberGloves with attached motion tracker to move a virtual hand in the VE. The user can use that hand to push objects that will behave in a physically realistic manner using the physics system.

```
audioSys->getStream("music")->play();

meshNode = gfxSys->getRootSceneNode()>createChild("robot");
Mesh* mesh = gfxSys->createMesh("robot", meshNode, me);
meshNode->setPosition(0, 0, 50);
mesh->setCastShadows(true);
mesh->getAnimation("walk")->setEnabled(true);

Sound* sound = audioSys->createSound("footsteps", meshNode,
"footsteps");
sound->setLoop(true);
sound->play();

lightNode = gfxSys->getRootSceneNode()->createChild("light0");
Light* light = gfxSys->createLight("light0", lightNode);
lightNode->setPosition(0, -50, 40);
```

Figure 5: Sample source code for creating a music stream, a mesh, a sound, and a light.

4.6 MAGE's Object-Oriented Design

Object-oriented programming is at the core of MAGE's design philosophy. The clean separation of different components in MAGE is made possible by the encapsulation of each system. Each system in MAGE has minimal coupling. This is possible because game engines are made up of heterogeneous systems that are tied together for the purposes of the game, but can exist on their own. This minimized coupling allows developers to safely swap out plugins without disturbing the state of the rest of the application. There are some interdependencies between systems, but MAGE is carefully designed so that the impact of the coupling is diminished. For example, the input system depends on the graphics system to provide a handle to the rendering window, so that it can capture mouse and keyboard events. MAGE handles this interdependency by having the input system request the window handle from the graphics system once on initialization, but the two systems never interact after that point.

The wrapping of each component into a plugin is accomplished through the use of inheritance and polymorphism, which are concepts from object-oriented programming. Inheritance is a method of creating classes by using the behavior of existing classes. Polymorphism is a method of varying object behavior based on the object's type. Each system and all the game objects the system can create are specified in abstract base classes that individual plugins must inherit and implement. The polymorphism is evident when each system displays different behavior depending on the underlying plugin that is loaded. The abstraction has many benefits to the developer using MAGE. Using the provided interfaces, any library can be integrated into MAGE. When developing an application with MAGE, the developer doesn't have to worry about such details as the different 3D vector and matrix formats of each library, since these details are abstracted out.

4.7 MAGE's Use of C++

MAGE, like most games and games engines today, is written in C++. An example of using MAGE source code can be seen in Figure 5. Although there are a growing number of projects using higher-level languages like Java and C# for game programming, almost all commercial games continue to use C++. The reason for this is that Java and C# support many features that improve programmer productivity at the expense of runtime performance. One good example of this is garbage collection. Garbage collection frees the programmer from cleaning up objects herself, but it can stall a game at inopportune times, which is unacceptable for applications that must meet soft real-time requirements.

It is possible, however, that game engines will transition to higher-level languages, just as the programming language of choice for game programmers has changed from assembly to C to C++. But since this requires substantially faster hardware, it will take at least several years for this to become a reality. In the meantime, a good approach to gain the benefits of high-level languages while retaining the performance of C++ is to use language bindings for scripting. Scripting has been used for years to allow non-programmers to design game logic and gameplay mechanics. Typically, only the most high-level functions of the game are made scriptable, therefore exposing only a little bit of functionality to the user. It is conceivable, however, that the entire game engine can have language bindings, therefore allowing the developer to write the game in their language of choice. For more information about this topic, see the "Future Work" section.

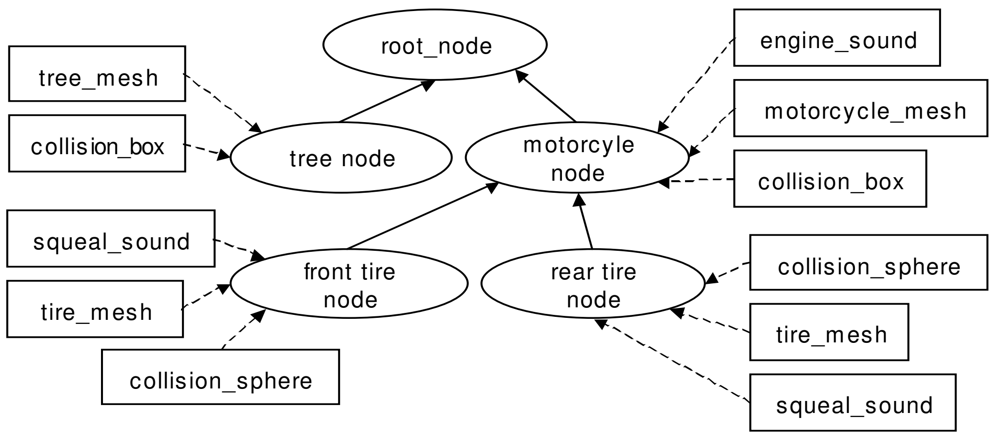


Figure 6: A diagram of a simple scene graph, with scene nodes in ovals and attachable objects in rectangles.

4.8 Scene Graph-Based Design

An important part of MAGE design is its handling of game objects. Game objects are discrete objects that can be moved in the 3D environment. Examples of game objects are 3D models, sounds, and collision proxies, which are shapes that estimate the boundary of an object. Game objects are created by systems, with plugins providing implementations for all of the game objects.

An unusual aspect of MAGE's design is that all the game objects, including the non-visual ones, are organized in a scene graph, which is a tree-like structure for creating a hierarchy of graphical objects (see Figure 6). Scene graphs are commonly used for graphics engines, but they are useful in game engines for two major reasons. Firstly, some things in real life can be described naturally in terms of hierarchies, such as a wheel to a car or branches to a tree. With this type of hierarchy, it is easy to move and rotate grouped objects by simply cascading changes in transform down the tree. For example, if we move a car from a parking lot to the highway, we want the tires to go with it. Child scene nodes can still move independently of their parent scene nodes, so that a tire could rotate when the car is turning, without causing the rest of the 3D car model to turn. The second reason to use scene graphs is to making frustum culling more efficient. Frustum culling is a method of removing objects that aren't in view. If a parent scene node is not in the view frustum, then both the parent and its children are not rendered (since the parent's bounding box encloses the children), therefore cutting down on processing time.

Scene graphs are typically used strictly as a method of organizing graphical structures. MAGE uses the scene graph for all other game objects because it helps both the spatial and logical organization of the game world. It helps spatially because some objects that should be moved together can just be attached to the same scene node. For example, if there is a man walking down the street while singing in the 3D environment, we can just create a scene node with a 3D model of the man with the sound of the song attached. Therefore as the scene node is moved down the street, both the model and the sound will move with it. The scene graph is also useful for logical organization because if two game objects should logically be grouped together, they can be attached to the same scene node. When that scene node is destroyed, both the game objects will be destroyed as well.

4.9 Focus on Rapid Application Development

MAGE has a strong focus on rapid application development for maximizing the efficiency of the programmer and minimizing the development time of the game or virtual reality application. This is accomplished by providing high-level functionality specifically targeted for game and virtual reality application developers. Little "shortcuts" are sprinkled throughout the library to save the developers from having to perform the tedious tasks themselves. There are many examples of this targeted functionality. One example is in the case when the developer wants to play a sound when two objects collide. This is a very common scenario for game developers, so MAGE simplifies the process by allowing the user to play sounds at the point of collision. This is contrasted with the tedious task of having to allocate the sound object manually, position the sound at the point of collision, play the sound, and then destroy the sound object.

4.10 MAGE's Behavior During a Typical Run

In this section, the behavior of a MAGE application during an average run will be described. During initialization, the user will specify which plugins are to be loaded for use. This can be either specified at compile-time or at run-time, using a simple configuration script. The order of loading plugins can be important if there are dependencies among them, so MAGE will automatically load certain plugin to fulfill dependencies, even if they are not explicitly specified by the user. As each plugin is loaded, the underlying libraries will perform their own initialization routines. For graphics libraries, this usually involves loading models, textures, and other resources from files, creating a render window with an appropriate pixel buffer, constructing a scene manager, and instantiating a camera.

After all the plugins have been loaded and the libraries have finished initializing, MAGE's main game loop begins. In this loop, the user has the chance to perform operations before all the various systems have been ticked for that frame by using the start frame callback. After the frame started signal has been sent, all the systems are updated the time passed since the last frame as the only parameter. The time is used for animation, stepping through the physics simulation, and constructing Doppler effects (the phenomenon one hears when an ambulance with its sirens on passes them) for the audio system. After the update signal has been sent, the end frame signal is fired, allowing the user to perform operations at the end of the frame.

When the application is ready to terminate, MAGE begins unloading plugins (in the reverse order of creating them) and destroys the instantiated libraries. Then MAGE frees up all the resources it has allocated itself and exits.

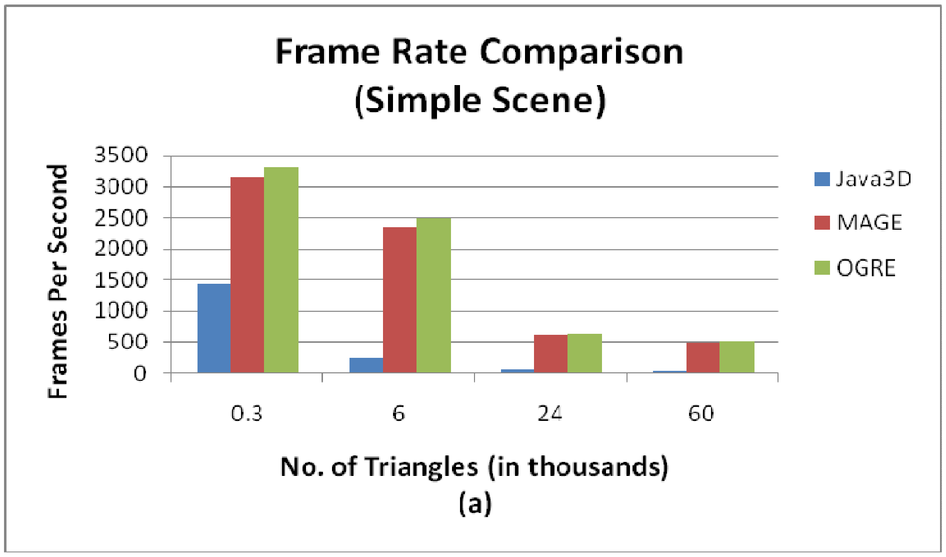
CHAPTER 5

PERFORMANCE EVALUATION

5.1 Performance Considerations

MAGE's architecture poses many interesting design issues, since MAGE's flexibility and power do not come without tradeoffs. The first and possibly the most obvious one is the issue of performance, because the increased levels of abstraction in a software system usually cause a decrease in performance. In MAGE, these performance penalties are manifested in several areas. The most important one is the additional overhead of extra virtual function calls. In game engines with either built-in systems or with fixed middleware components, the function calls to the systems can be made directly and efficiently. By contrast, MAGE must make polymorphic function calls, which have overhead of virtual function tables and late-binding [22]. A virtual function table is a data structure used to store all the virtual functions present in a given class hierarchy. Late-binding is the concept of associating an object with its type at runtime.

Another sacrifice in performance MAGE makes for the sake of flexibility is the message passing and type conversions between systems. One common example of this occurrence is when the user decides to move a scene node from one position to another. This will necessitate the notification of the underlying graphics engine of the change and the conversion of the 3D vector structure from the MAGE format to one the graphics engine can understand. If the scene node also contains a 3D sound (a sound with a three dimensional position and rotation), the audio system must in turn be notified and the vector then needs to be converted to the audio system's format, and so on. Although these type conversions can be made relatively efficient by inlining the function calls or by writing them in assembly, they are frequent enough to cause slowdowns in the client application. These type conversions would not be necessary if the system components were integrated into the main game engine.



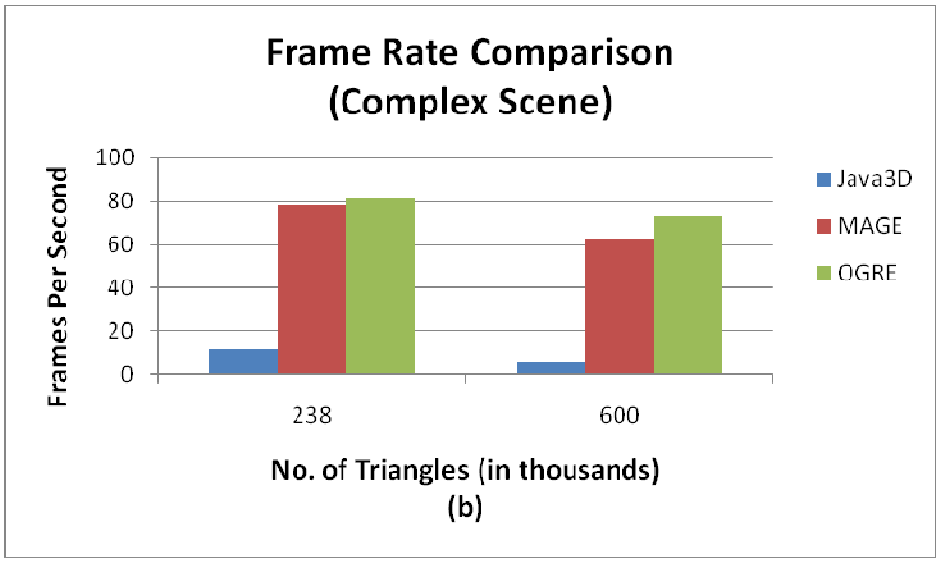


Figure 7: Comparison of frame rates for Java3D, MAGE-VR, and OGRE for (a) simple scenes and (b) complex scenes.

5.2 Results of Performance Evaluation

Performance issues are of utmost importance to games and virtual reality applications since these applications must, by definition, run at interactive speeds. However, performance cannot be deduced by simply analyzing the design of the software system. The performance of the software must be measured, tested, and verified.

In this section, we describe the results of our tests comparing the frame rates of identical scenes rendered in MAGE, OGRE, and Java3D [20] (see the MAGE-VR paper for more details on these results [4]). The purpose of these tests is to demonstrate how much overhead MAGE adds to the underlying rendering components and how MAGE compares to a high-level graphics package written in Java. We measure overhead by comparing a version of MAGE using OGRE as the active rendering plugin to OGRE running by itself. Since OGRE only handles graphics and basic input, MAGE incurs a small performance penalty from adding an extra layer of abstraction, but allows it to handle VR input devices, audio, collision detection, and physics in a single integrated framework. Finally, we evaluate MAGE to Java3D to compare MAGE's performance to a high-level graphics package.

Frame rates are affected by other applications and operating system services running on the test machine. For this reason, frame rates represent an estimation of performance of the libraries being tested. Furthermore, graphics are just one part of an interactive system that can cause latency, but they are indicative of the relative performance between MAGE and Java3D.

The test machine used for experimentations was a single-processor 3.2 GHz Pentium 4 with 1 gigabyte of RAM and a 256 MB Geforce 6800 graphics card. The operating system used was Windows XP. For the MAGE tests, OGRE was used as the underlying graphics library and all other plugin systems were disabled. For Java3D, the JWSU toolkit was used for handling basic scene creation and input handling [24]. For each test, the camera was fixed on one or more instances of a single model. The model was loaded in the OGRE mesh file format for MAGE and OGRE and in VRML for Java3D. DirectX was used for all tests and the graphics are rendered in a 640 x 480 window with 32-bit color.

Figure 8: A model made up of 6,000 triangles that was used in the tests. (Model courtesy of De Espona Infografica)

The first four tests, shown in Figure 7a, compared the frame rates obtained while rendering relatively simple scenes. One of the models used for the test is shown in Figure 8. In the first test, the scene was composed of one simple model made up of 308 triangles. In this test, MAGE's frame rates are 2.11 times that of Java3D's. MAGE added a modest overhead of 7% compared to OGRE's frame rates. The second test, with approximately 3000 triangles, showed MAGE's frame rates to be 7.88 times that of Java3D's frame rate and with 4% overhead compared to OGRE by itself. The third test with about 24,000 triangles showed MAGE 7.51

times the frame rate of Java3D and MAGE's frame rate was 0.4% below OGRE's. The final simple test was a scene with 60,000 triangles and MAGE maintained 15.49 times the frame rate as Java3D with an 8.4% overhead compared to OGRE.

The final two tests, shown in Figure 7b, demonstrated complex scenes that have a large number of triangles on screen at once. In the first test, a scene with approximately 238,000 triangles on screen at once was used. The test showed MAGE having 7.78 times the frame rate as Java3D with an overhead of less than 1%. The final stress test was a scene with 600,000 triangles, and MAGE obtained 16.70 times the frame rate as Java3D with an overhead of 9.5%.

Our comparisons show that MAGE outperforms Java3D in both simple and complex scenes and incurs little overhead over just using OGRE graphics. Furthermore, the tests show that MAGE generally scales better than Java3D as the scene gets more complex. This is an important factor that will allow game and VR application developers using MAGE to implement highly detailed worlds while still maintaining high frame rates.

5.3 Analysis of Performance Results

From the performance results, MAGE adds little overhead to its underlying libraries, outperforms an engine with similar features, and vastly outperforms Java3D. The reason MAGE is able to reach this level of performance is because its flexible architecture allows the developer to choose the highest performing libraries instead of being locked into one type of system. Even though this flexibility adds some overhead, it is evident from the OGRE and MAGE comparisons that this overhead is minimal. The reason for this is because the computationally expensive operations in rendering far outweigh the cost of the extra layer of abstraction.

CHAPTER 6

APPLICATIONS USING MAGE

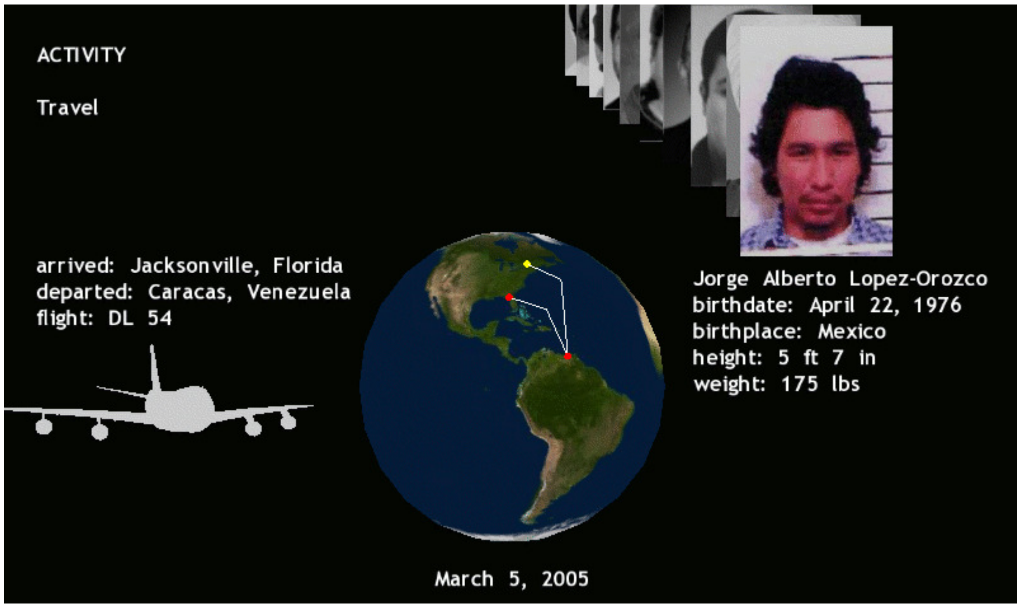

ACTIVITY
Travel
arrived: Jacksonville, Florida
departed: Caracas, Venezuela
flight: DL 54
Jorge Alberto Lopez-Orozco
birthdate: April 22, 1976
birthplace: Mexico
height: 5 ft 7 in
weight: 175 lbs
March 5, 2005

Figure 9: A screenshot of SemanticSpy.

6.1 SemanticSpy

SemanticSpy [3] is tool for visualizing the activities of persons of interest around the world in a multimedia environment using semantic data. SemanticSpy is intended for use by anti-terror and law enforcement agencies for tracking suspects as they travel and perform activities such as making phone calls or attending meetings. By visualizing the movement and habits of various individuals, the user may be able to notice suspicious patterns that they may not otherwise notice if they were viewing the data as just text.

SemanticSpy demonstrates the use of MAGE for providing a multimedia environment by displaying three-dimensional models, playing audio files, and providing interactivity for the user. A screenshot of SemanticSpy is shown in Figure 9. The three-dimensional models are used for displaying an interactive globe that can be rotated to show the travel routes of a suspect. Models are also used to show the modes of transportation used by the suspect, such as the model of the car or plane the suspect is traveling in. This may provide actionable information by law enforcement agencies if, for example, they see a suspect traveling in the same plane type on several occasions. This may indicate that the suspect is planning an attack on board such a plane in the future.

SemanticSpy also uses audio to convey pertinent information to the user. The most important use of audio is for playing back phone calls or other conversations of the suspects. Phone calls may contain important bits of information that may lead to clues about nefarious plans the suspect may be planning.

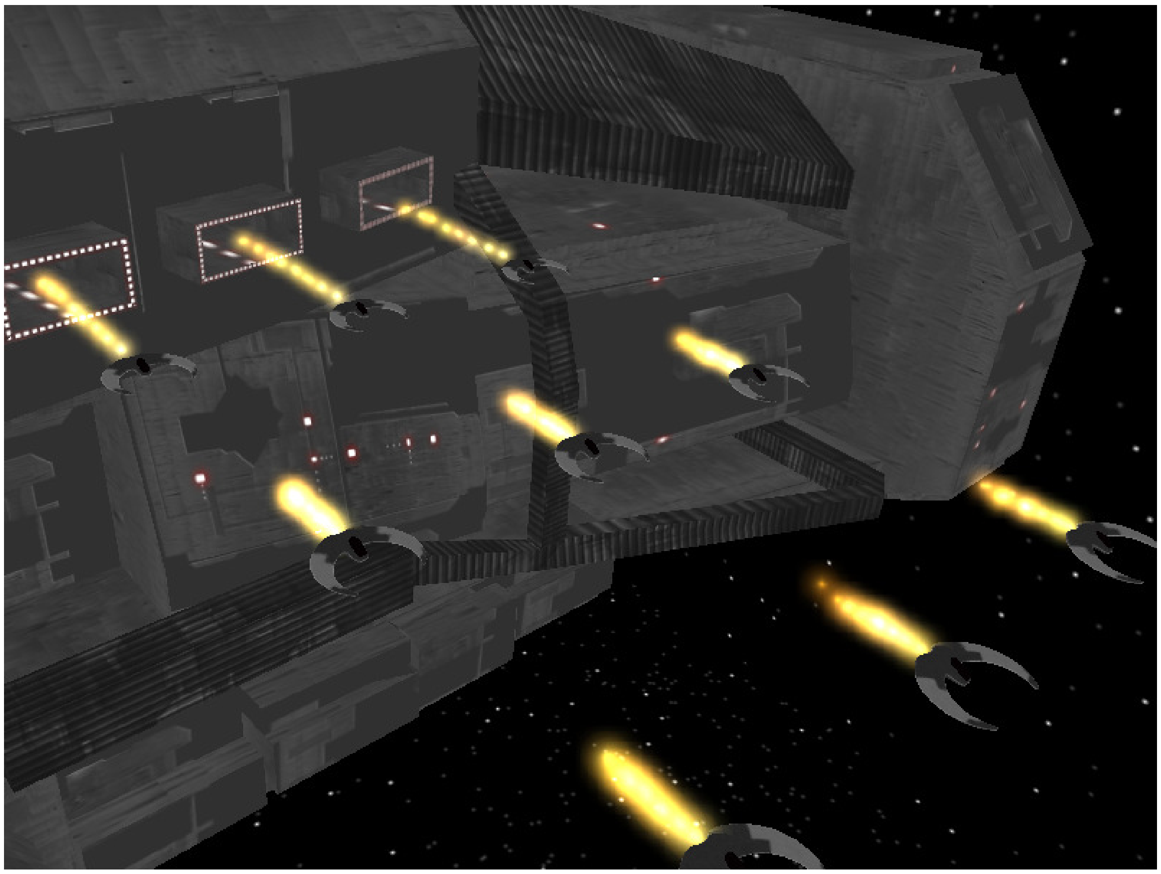


Figure 10: An early screenshot of Battlecrest.

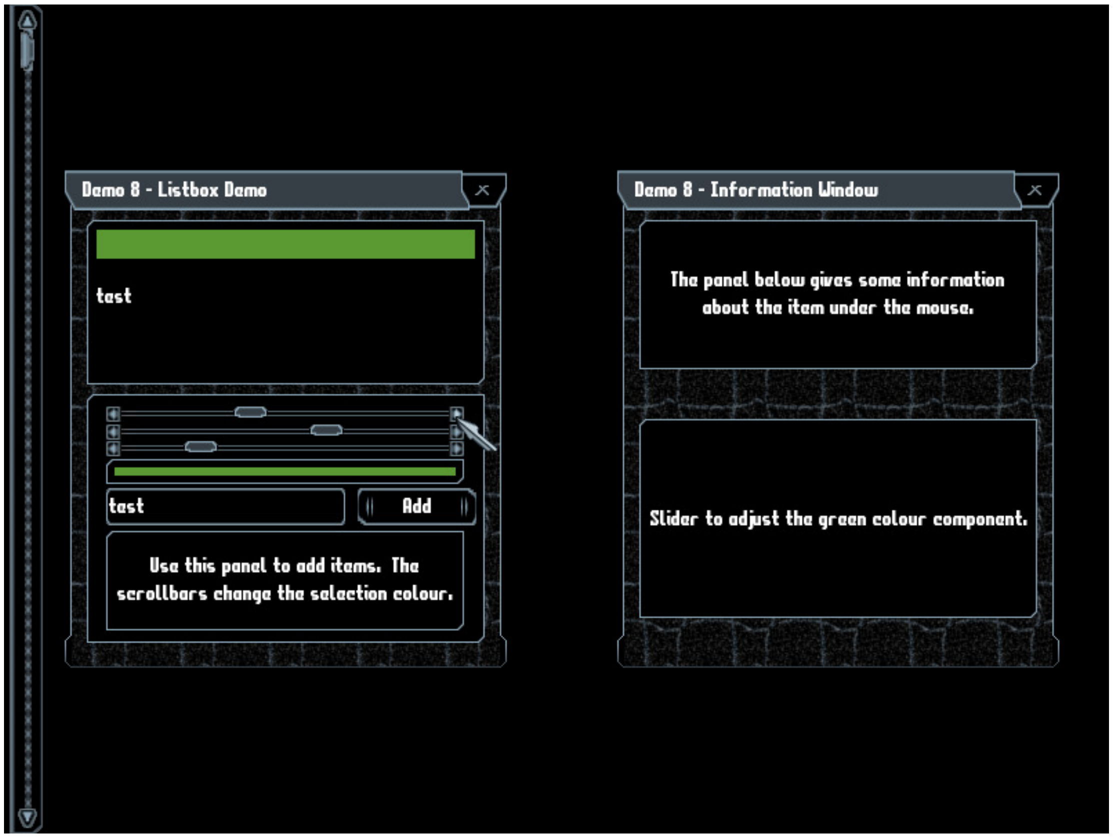
6.2 The Battlecrest Project

Battlecrest, shown in Figure 10, is a space combat game with a focus on realism that is currently in early stages of development. The project is meant to be a proof-of-concept that a commercial-quality game can be developed using MAGE. The game will make extensive use of physics to simulate spacecraft operating in a zero gravity environment.

CHAPTER 7

FUTURE WORK AND CONCLUSION

MAGE is presently not a complete engine. Several features must be implemented before it will be comparable to commercial-quality engines. MAGE requires extensive development to extend existing systems and to add crucial new ones. Fortunately, many of these missing features are either currently in development or will be developed in the near future.



Use this panel to add items. The scrollbars change the selection colour.

The panel below gives some information about the item under the mouse.

Slider to adjust the green colour component.

Figure 11: Crazy Eddie's GUI running through OpenGL.

7.1 Graphical User Interface System

One of the systems in development is the graphical user interface (GUI) system. GUIs in 3D environment are difficult to integrate into MAGE because they are usually tied to the graphics engine being used, which contradicts MAGE's design philosophy of reducing the dependencies between systems. This can be overcome by using a GUI system that is not dependent on any single render system. There are very few of these systems available, the most prominent probably being Crazy Eddie's GUI System [6], show in Figure 11. The difficulty in integrating such a GUI system is creating an abstract interface to a GUI system, which have many different functions. One approach to tackle this problem is by using data files, which is described in Section 3.3.

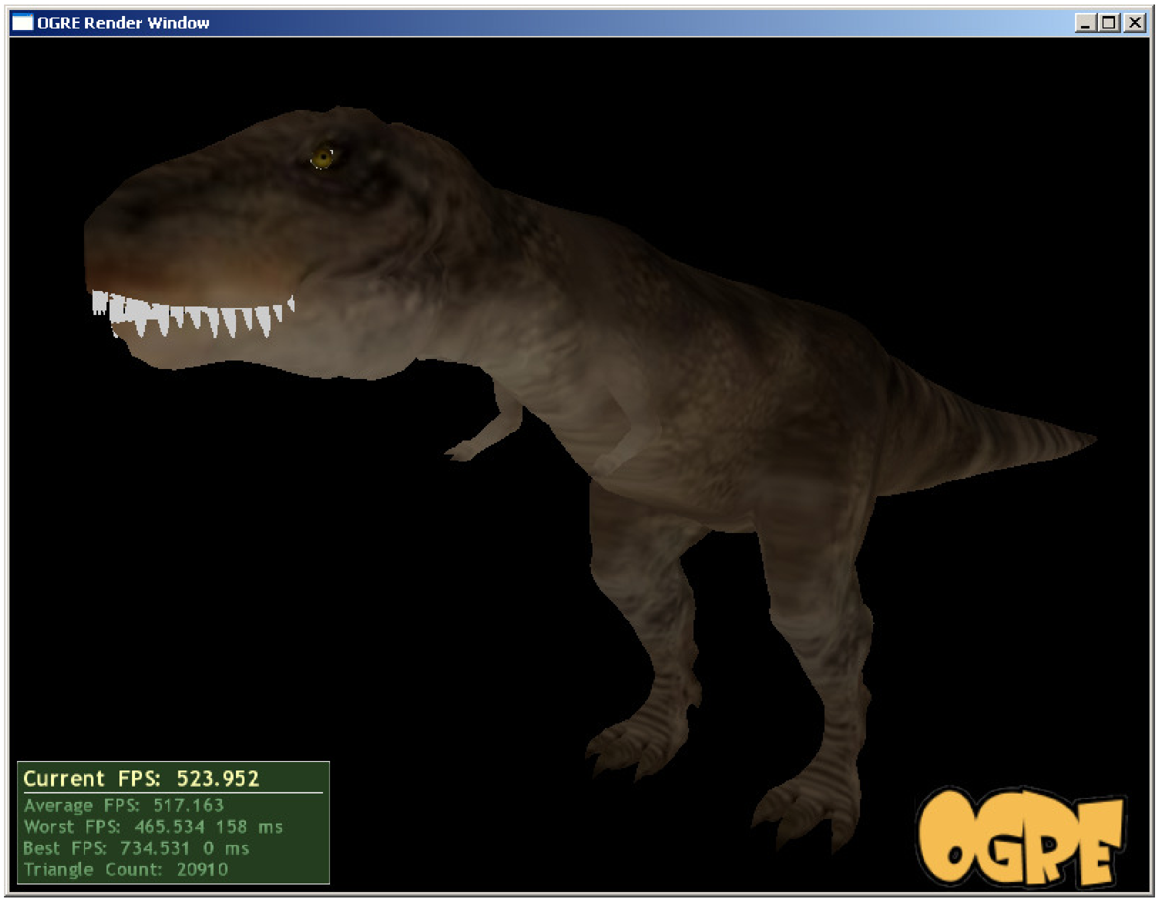
7.2 Game Artificial Intelligence System

Another important system that is absent from MAGE is artificial intelligence (AI). AI in the context of game development is actually a narrow subset of the field of AI. More about game AI can be found in [1], [27], and [41]. One of the challenges of integrating AI into MAGE are that AI systems are usually very game-specific and MAGE is intended to be a general purpose engine. Also there is a dearth of free general-purpose game AI systems, so one will probably be built from scratch for MAGE. Commercial options do exist, however, like the Renderware AI system [38]. There are options for specific areas of game AI, like pathfinding, where OpenSteer [35] is a good example.

7.3 Language Binding and Scripting

A language binding is a way of allowing the programmer to use one language to call methods for a software system written in a different, usually lower-level language. This allows the programmer to use a high-level scripting language to reduce development time while still maintaining the performance characteristics of the underlying software system. Panda3D is an excellent example of using language binding in practice. Although the core Panda3D engine is written in C++, the engine has Python [15] language bindings, allowing the programmer to develop Panda3D applications with Python. This allows the programmer to rapidly prototype games using Python, without needing to recompile the game every time a change is made, since Python is an interpreted language. Since the rendering and other computationally expensive operations are still done in the core Panda3D library, the game achieves acceptable performance.

Scripting is the technique of using language bindings for very high-level game logic, allowing game designers without much experience in programming to develop their own scripts. Scripting is used extensively in modern games, with the Half-Life [16] series of games being a famous example. Scripting is an important part of the content pipeline, which is discussed below.



```
Current FPS: 523.952
Average FPS: 517.163
Worst FPS: 465.534 158 ms
Best FPS: 734.531 0 ms
Triangle Count: 20910
```

Figure 12: A 3D model exported to the OGRE format from the FBX format.

7.4 Tools for the Content Pipeline

The game engine is just one part of developing a complete game. The artistic content is what completes the game. This content includes assets like levels, 3D models, textures, shaders, GUI elements, fonts, music files, sound effects, and game scripts. Importing these assets from the digital content creation tools into the game engine is a significant task. To do this, MAGE relies on individual libraries to provide exporters for popular formats in their domain. OGRE provides exporters for many major 3D modeling programs and Irrlicht supports a few popular model formats, including the OGRE format. The audio libraries support loading .wav and .ogg files. Standardized physics formats are still in their infancy and physics libraries have varying levels of support for them, but they may be an important feature in the future.

Just being able to use the assets in the game engine is not sufficient, however. Artists and designers must be able to see their changes made in their tools represented in the engine as quickly as possible, or it will quickly become a bottleneck in the game development process. There are various techniques to accomplish this task. One is to provide tools that allow designers to view different types of content in a single application. For example, if there is a scene in the game where an armored car crashes into a bank, the designer would want to see the models of the car and the bank, hear the sounds of the crash, and view the physics of the impact.

The process of exporting artistic assets from the tools in which they were created and allowing these assets to be viewed and edited quickly in the game is called the content pipeline. Optimizing this content pipeline is an important task for game development teams. MAGE currently relies on the libraries themselves for basic content pipeline functionality, but in the future will provide more robust tools. One early step in this goal is the FBX exporter (see Figure 12), which exports the FBX animation format [12] to the OGRE mesh format.

7.5 Networking System

Networking is an important feature for today's online multiplayer games. In the future, a networking system will be added to MAGE that supports the automatic replication of game objects to multiple clients of the network. The goal is to make adding multiplayer capabilities to developers using MAGE as transparent as possible.

7.6 Portability

Currently MAGE only builds under Microsoft Windows. Making MAGE portable across different platforms is an important task. Although a significant amount of work is required to realize the goal, it is a manageable problem since all the underlying libraries used by MAGE are cross-platform. This means just the MAGE core needs to be ported to other platforms. Furthermore, MAGE's interaction with the underlying OS and hardware is minimal and is currently confined to the resource management system.

7.7 Conclusion

MAGE's plugin-based architecture offers developers flexibility and power by allowing them to choose the libraries that fit their needs. By choosing the high-performance libraries, MAGE can maintain excellent frame rates, even with very complex scenes. This is because MAGE has a very low overhead and most of the frame time is spent in the underlying libraries.

This flexible architecture allows MAGE to be used for a variety of different games types. Through MAGE-VR, MAGE can also support virtual reality devices with few modifications to the core library.

REFERENCES

[1] Alexander Nareyek, “Intelligent Agents for Computer Games.” In Marsland, T. A., and Frank, I. (eds.), Computers and Games, Second International Conference, CG 2000, Springer LNCS 2063, 414-422.

[2] Allen Bierbaum, Christopher Just, Patrick Hartling, Kevin Meinert, Albert Baker, and Carolina Cruz-Neira, “VR Juggler: A Virtual Platform for Virtual Reality Application Development.” IEEE VR 2001, Yokohama, Japan, March 2001.

[3] Amit Mathew, Amit Sheth, and Leonidas Deligiannidis, "SemanticSpy: Suspect Tracking Using Semantic Data in a Multimedia Environment." In. Proc. of the IEEE International Conference on Intelligence and Security Informatics (ISI-2006), May 23-24 2006, San Diego, CA, 492-497.

[4] Amit Mathew and Leonidas Deligiannidis, “MAGE-VR: A Software Framework for Virtual Reality Application Development.” Proc. of the 2005 International Conference on Modeling, Simulation and Visualization Methods (MSV'05), June 2005, 191-197.

[5] *Asteroids.* Video Game. Atari Corporation, 1979.

[6] Crazy Eddie’s GUI System, http://www.cegui.co.uk/, 2004.

[7] CyberGlove product, Immersion Corporation, 2008.

[8] David Eberly, *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic*. Morgan Kaufmann Publishers Inc. 2004.

[9] Microsoft DirectX. Microsoft Corporation, 2008.

[10] *Doom*. Video Game. Id Software, 1993.

[11] Doom 3 Engine. Id Software, 2003.

[12] FBX animation format, Autodesk, http://www.autodesk.com/fbx/, 2006.

[13] Fmod, Firelight Technologies, http://www.fmod.org, 2006.

[14] Gamebryo, Emergent Game Technologies, http://www.emergent.net/, 2008.

[15] Guido Van Rossum, *Python Tutorial*, Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

[16] *Half-Life*. Video Game. Valve Software, 1998.

[17] Havok. Havok.com Inc. 2008.

[18] Infocom’s Z-machine, Activision, 1979.

[19] Irrlicht Engine, http://irrlicht.sourceforge.net, 2005.

[20] Java3D, http://java.sun.com/products/java-media/3D/, 2006.

[21] Jeff Plummer, “A Flexible and Expandable Architecture for Computer Games.” Master’s Thesis. Arizona State University. 2004.

[22] Karel Driesen, Urs Hölzle, “The direct cost of virtual function calls in C++.” Proc. of the 11th ACM SIGPLAN Conference on OOPSLA, 1996, 306-323.

[23] Larry Bishop, et al, “Designing a PC Game Engine.” Computer Graphics and Applications, IEEE. Jan/Feb 1998, 18:1, 46-53.

[24] Leonidas Deligiannidis, Gamal Weheba, Krishna Krishnan, and Michael Jorgensen, "JWSU: A Java3D Framework for Virtual Reality". Proc. of the International Conference on Imaging Science, Systems, and Technology (CISST03). June 2003, 312-319.

[25] Mark Segal, Kurt Akeley, “The Design of the OpenGL Graphics Interface,” Silicon Graphics, 1994.

[26] Markus Hadwiger, “Design and architecture of a portable and extensible multiplayer 3d game engine.” Master’s Thesis, Institute of Computer Graphics, Vienna University of Technology, 2000.

[27] Michael Van Lent et al, “Intelligent Agents in Computer Games,” Proceedings of the National Conference on Artificial Intelligence. Orlando, FL, July 1999, 929-930.

[28] Mike Goslin, Mark R. Mine, "The Panda3D Graphics Engine," Computer, vol. 37, no. 10, Oct 2004, 112-114.

[29] Newton Game Dynamics, http://www.newtondynamics.com/, 2006.

[30] Open Dynamics Engine (ODE), http://www.ode.org/, 2001.

[31] OGRE, http://www.ogre3d.org/, 2006.

[32] OIS, http://sourceforge.net/projects/wgois/, 2006.

[33] OPAL, http://opal.sourceforge.net/, 2005.

[34] OpenAL, http://www.openal.org/, Creative Labs, 2006.

[35] OpenSteer, http://opensteer.sourceforge.net/, 2004.

[36] Polhemus Liberty, Motion tracker. Polhemus, http://www.polhemus.com/, 2006.

[37] *Pong*. Video Game. Atari Corporation, 1972.

[38] RenderWare, Criterion Software, 2008.

[39] Script Creation Utility for Maniac Mansion (SCUMM), LucasArts, 1987.

[40] SpeedTree, Interactive Data Visualization Inc., http://www.speedtree.com/, 2008.

[41] Steven Woodcock, "Game AI: The State of the Industry" in *Game Developer Magazine*, August 2001.

[42] Unreal Engine, Epic Games Inc, http://www.unrealtechnology.com/, 2008.