Development and Deployment of Cross-Platform 3D Web-based Games

Chris Carter, Abdennour El Rhalibi, Madjid Merabti
School of Computing & Mathematical Sciences
Liverpool John Moores University
Liverpool, UK
{C.J.Carter@2007.}{A.Elrhalibi@}{M.Merabti@} ljmu.ac.uk

ABSTRACT

Digital distribution is becoming an increasingly important technique within the games industry. The leading consoles each possess their own bespoke platform to digitally deploy game applications to their users via the Internet, whilst the Windows and Mac OS X gaming market is catered for by systems such as Valve's Steam platform. However, these digital content services are often machine-specific, proprietary systems, utilising custom web frameworks and a rigid publication system. In this paper we present an overview of the Homura games framework and Net Homura deployment middleware. Homura provides an integrated solution for Java based 3D games development. We also discuss a novel architecture and prototype system, which aims to unify the deployment of hardware-accelerated Java-based 3D games applications with online capabilities. Net Homura provides a multi-tiered deployment platform that is secure, robust, and easily portable to a wide range of web servers; whilst the networking middleware component of Net Homura allows developers to build content and feature rich online games in conjunction with the Homura Engine and IDE. Due to nature of the technologies used within the Net Homura framework, it is possible to enable the creation of a game which, from development through to hosting, deployment and networking, can be created with little or no financial outlay for the developer by utilising open source technologies.

Keywords: Homura, Net Homura, Java, jME, Java Web Start, Applets, Deployment, Digital Distribution, Game Engine, Web Games.

1. INTRODUCTION

The introduction of the "next-generation" console systems has seen an increasing focus on the digital distribution of software. Each console has its own platform-specific digital content distribution mechanism – Nintendo Wii has the Wii Shop Channel [1], the Sony PS3 has the Playstation Store [2] and Microsoft has the Xbox 360 Live Marketplace [3]. In each case, store front-end applications are embedded into their console's Operating System as bespoke platform-dependent applications and utilise the internet connectivity which each machine possesses to provide their consumers easy access to the online distribution stores. Each platform provides similar functionality: A browsable catalogue of downloadable content; A mechanism to download and install the content locally onto the games console; and a variety of content types including full games, retro game emulations, game add-ons, game demos etc. The deployment of games software is not limited to the console platforms and has become a growing market for PC and Mobile games. Valve's Steam distribution platform [4] supplies over 600 PC titles and has over 20 million registered account holders, whilst Apple's AppStore [5] system for the iPhone has gained the support and the release of titles from major developers and publishers such as EA (Sims 3) and PopCap Games (Peggle). However, a major problem is that each of these digital distribution systems is platform-specific and proprietary. They are all also reliant on bespoke client applications (e.g. iTunes, the Steam client). Therefore, this paper presents an open-source platform for the development and deployment (digital distribution) of modern game applications, which can be both distributed and executed in a consistent cross-platform, cross-browser manner: the Homura project [6]. Section 2 of this paper provides a discussion of the work related to the production of the Homura framework. Section 3 provides a detailed technical overview of the two main constituents of the Homura project and the prototypes developed to test the concepts and interoperability of both systems. Section 4 analyses the proposed deployment solution. Finally, in Section 5 we conclude the paper and discuss future work.

2. RELATED WORK

In order to design the Homura framework, a detailed look into three related aspects was required: Existing engines and frameworks related to development and deployment of games via the Internet. We also appraised various programming languages and technologies to determine their suitability for both web-based and games development, resulting in the choice of Java as our development language; subsequently, in this section, we provide an overview of the deployment techniques available for Java. Finally, we needed to determine features which are necessary for an open games development platform.

2.1 Existing Technologies

There are two primary frameworks which support both the development and digital deployment of games applications: Unity [7] and Microsoft's XNA Game Studio [8]. Both of these are proprietary solutions with closed source APIs. Unity supports both Windows and Mac OS X, through its custom browser plug-in, the Unity Web Player [7]. It requires a different plug-in for each supported platform (such as the ActiveX control for Windows Internet Explorer). Unity applications are primarily developed graphically using its custom development environment and scripted using Mono (an open source .NET implementation), which supports C#, Boo and JavaScript as the development languages. Unity has many different license models from independent to professional level including, since October 2009, a free edition. XNA is a games development framework which provides a managed run-time environment for the development of computer games using the .NET 2.0 frameworks. XNA is mainly used with C# as the development language, and is available as integration for the Visual Studio development environment (professional and express editions). XNA supports development for both Windows PC and Xbox 360. To distribute games on this platform a peer review system must be passed and a yearly development subscription of $99 is required, with developers receiving 70% of the revenue of their creations [8].

2.2 Java-Based Deployment

The Java language, its execution environment, and the suite of core APIs and classes together provide facilities that recommend it for use in implementing systems distribution. Its ability for the same codebase to run across different platforms, its safety features and its support through the class loader system for dynamically incorporating new code makes it particularly suitable for systems where behaviour and configuration are expected to change over time. Since update 10 of Java Standard Environment (SE) 6, there are two mechanisms for the deployment of Java applications: Java Web Start (JWS) [9] and next-generation applets; both are components of the new Java Plugin v2, which is distributed as a part of the Java Runtime Environment (JRE). The Java Plugin is freely available for all major operating systems and browser environments, making the technology ubiquitous amongst desktop PC users. Next generation applets are a major upgrade and have been modified to have architectural parity with JWS. Applets are now executed outside the browser as a separate process which is controlled by a lightweight, headless virtual machine (JVM) which sits inside the browser [10]. Both the Applet and Web Start technologies utilise the Java Network Launching Protocol (JNLP) [11] to configure exactly how an application is deployed from a server location to the clients' machines. The protocol uses a standardised XML schema to define several key aspects of the deployment process, such as application skinning; defining the libraries which comprise the application; OS and architecture (x86/x64) specific libraries; security permissions etc. Sections within the schema can be used to define properties explicitly for either JWS or Applets, such as the main application class name and the Applet implementation. Homura has been designed to support distribution using both next-generation applets and Web Starts. Java is already being used as a web distribution platform for simple 2D game applications via the browser, such as EA's casual games platform known as Pogo [12].

2.3 Design Criteria and Rationale

There are several key issues that need to be addressed, and desirable features which need to be implemented when undertaking the development of an open, web-based deployment platform for games applications:

- **Security:** The applications must be able to be delivered in a secure manner, allowing the integration of authentication and authorisation mechanisms, as well as validation method to ensure the authenticity of the application to the user.
- **Integration with existing web technologies:** In order to maximise the accessibility of the platform, it should easily allow the integration of the games application with a variety of common web application frameworks.
- **Cross Platform Consistency:** To maximise the user base and minimise the development work required to execute / port the games across multiple hardware configurations and operating systems.
- **Cross Browser Consistency:** The games should be interoperable with the most common browsers available on a given platform (IE 6/7/8, Firefox 3, Google Chrome, Safari etc) in a standards compliant, consistent manner, which does not require hacks in order to correctly support each program.
- **Application Updates:** An easy mechanism for updating the versions of the application must be made, so that consistency amongst clients can be ensured and patches can be made to eradicate bugs etc.
- **Download Size:** The download size of the application must be as small as possible to minimise the bandwidth usage and perform adequately on slower connections. Support for techniques such as compression and caching should be provided.
- **Application Performance:** The games should be able to make use of the processing power and hardware capabilities that a modern desktop PC possesses. Utilisation of hardware acceleration and modern features such as programmable pipeline rendering should be handled by the framework.

3. SYSTEM OVERVIEW

The Homura project [6] is comprised of three distinct sub-projects that are designed to interoperate with each other in order to provide a consistent platform for the development of web-deployable games applications: *Homura***,** an application framework for the development of hardware-accelerated 3D games using Java and OpenGL. Homura provides a vast array of functions and solutions to common game development techniques - as detailed in section 3.1; *Net Homura,* a web-based Framework for the development of websites / web applications which constructs pages in a fashion analogous to Homura scene construction using Object-Oriented PHP / HTML / CSS / JavaScript. It allows the seamless integration of Homura games into web pages - as detailed in section 3.2; *Homura IDE***,** the Homura IDE project, detailed in [13], aims to provide a game-oriented development environment built on top of the Eclipse IDE for the creation of Homura games. The IDE project is outside the scope of this paper, and will not be covered in further detail.

3.1 Homura – The Games Framework

The Homura framework is an Application Programming Interface (API) which aims to provide an open-source platform for the development of hardware-accelerated 3D games in Java. This section describes the application architecture, core feature set and key information regarding the implementation of the core classes which comprise the API [6].

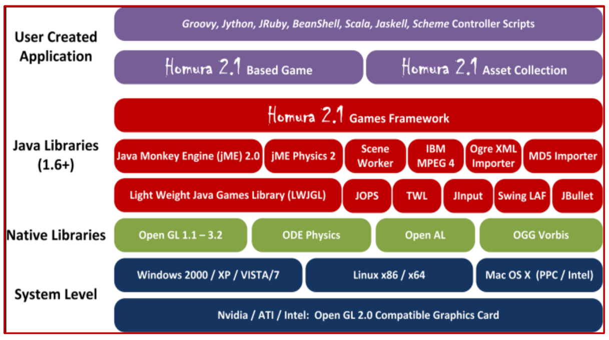

User Created Application
Groovy, Jython, JRuby, BeanShell, Scala, Jaskell, Scheme Controller Scripts
Homura 2.1 Based Game
Homura 2.1 Asset Collection
Java Libraries (1.6+)
Homura 2.1 Games Framework
Java Monkey Engine (jME) 2.0
jME Physics 2
Scene Worker
IBM MPEG 4
Ogre XML Importer
MD5 Importer
Light Weight Java Games Library (LWJGL)
JOPS
TWL
JInput
Swing LAF
JBullet
Native Libraries
Open GL 1.1 – 3.2
ODE Physics
Open AL
OGG Vorbis
System Level
Windows 2000 / XP / VISTA/7
Linux x86 / x64
Mac OS X (PPC / Intel)
Nvidia / ATI / Intel: Open GL 2.0 Compatible Graphics Card

Figure 1: Homura Application Architecture

3.1.1 Application Architecture

Modern game applications are becoming increasingly complex and are typically comprised of several interoperable sub-systems, each handling an aspect of the game such as two-dimensional and three-dimensional rendering, physics simulation, particle effect systems, audio, input-device control, Artificial Intelligence etc. The Homura games framework utilises many open-source libraries to build a powerful Java-based API to allow developers to easily construct their game applications by unifying these sub-systems into a single library. Figure 1 illustrates the typical architecture of a game application built using Homura.

The bottom layer of the architectural stack is the *System* layer. Homura is a cross-platform framework and will run on Windows, Linux and Mac OS X with the requirement of an OpenGL compatible graphics card. The second layer is the *Native Libraries*; Homura is written in Java but utilises native, platform-specific libraries for the key sub-systems in order to provide the best combination of performance and feature support, by allowing hardware-accelerated rendering and audio to be utilised. Homura relies on the native versions of OpenGL for rendering support, Open Dynamic Engine (ODE) for Physics simulation, OpenAL for Audio support and OGG Vorbis for open source multimedia format support. Java interfaces with these libraries using the Java Native Interface (JNI). The Homura Framework comprises the topmost layer of the API and is programmed exclusively in *Java*. All libraries directly referenced by Homura are also Java based, with these libraries handling the calls to the Native libraries. This approach was chosen because these existing libraries are already established and have been optimised to handle the native calls in the most efficient way, whereas Homura is primarily concerned with the high-level architecture of a games application. Homura utilises a custom variant of the Java Monkey Engine (jME) to provide rendering and input handling functionality. jME is an open-source technology which, over the last five years, has matured into a feature rich system which is one of the most performant graphical implementations in Java for 3D applications. jME uses the Lightweight Java Games Library (LWJGL) as its low-level OpenGL-based rendering sub-system. The primary function of LWJGL is to act as a Java binding to OpenGL by mirroring the interface of the C OpenGL library with a Java version of each function. For example, OpenGL's `glBegin()` is adapted as `GL11.glBegin()` in LWJGL. The LWJGL function will then utilise Java's JNI system to call the native version of `glBegin()`, and uses Java's NIO system to pass information between OpenGL and LWJGL as ByteBuffers, jME also provides a high performance scenegraph based graphics API; the scenegraph allows the organization of 3D geometry into a tree-like structure, where a parent *node* can contain any number of child nodes, but a child node must only belong to a single parent. The nodes are organized spatially so that whole branches of the graph can be culled when outside the view frustum. This allows for complex scenes to be rendered quickly, as typically, most of the scene is not visible at any one time. The scenegraph's *leaf nodes* consist of the geometry that will be rendered to the display.

Homura's audio sub-system again relies on LWJGL to provide the native bridge to the OpenAL audio library. Homura also utilises a jME sub-project, jME Physics 2 to provide the physics simulation functionality of the framework. This library integrates tightly with the jME scenegraph by virtue of its object classes inheriting from the jME scenegraph *Node* class. The physics subsystem uses the concept of *Static* and *Dynamic* node types; Static nodes are nodes that are not affected by physics, but other objects still can react physically to them (e.g. a wall), Dynamic nodes can be affected by forces and mass such as gravity and collisions with other physics objects (e.g. modelling a bouncing ball colliding with the static wall). JNI is used to bridge jME Physics with ODE to provide the low-level physics functionality. Another library utilised by Homura is the Java Open Particle System (JOPS), a framework which allows the creation of advanced particle effects (Smoke plumes, explosions, fireworks etc.) designed for LWJGL. This has been integrated into Homura by incorporating the JOPS file type into the Homura asset management system and encapsulating the particle generators as a specialised scenegraph node (a *JOPSNode*) which facilitates integration directly into the scene, or attached to a game entity node (e.g. the exhaust of a car for exhaust fumes). The framework composites a large set of disparate components into a single system, allowing a game to be easily built on top of the Homura system through linkage with the project's binary Java Archive (JAR) file.

Consequently, the final architectural layer is the *User-Creation* layer, which comprises the developed game. A game inherits from the Homura base classes (as described in 3.1.3) to provide the skeleton game: complete with all the aforementioned sub-systems. These classes implement the required game logic and the user-developed content (Meshes, textures, particle effects, audio etc) which are stored as a Homura asset collection and loaded through the game engine pipeline, using the Homura asset loader to construct the virtual environment which embodies the game. Whilst the core of a Homura-based game is developed in Java, non-performance critical sections of the game (e.g. some parts of the game logic) can be implemented as Scripts. Homura supports a variety of languages including Scala, Jython, JRuby and JavaScript. Scripts have access to the full Java API and can easily control any portion of the scene-graph and are suitable for a variety of purposes such as AI, cinematics, animation control, event triggers etc.

3.1.2 Application Partitioning

The previous section detailed the functionality provided by Homura and the architecture underpinning the framework. This section aims to provide an overview of how Homura can be used to create games applications which can be deployed in a cross-platform, cross-browser manner. In order to achieve this, a partitioning had to be made to separate the game from its underlying display context (browser, application window, full-screen mode) abstracting the display system. As a result of this partitioning, there are three roles that need to be fulfilled by any Homura-based game: Executor, Instantiator, and Controller. Figure 2 illustrates this partitioning.

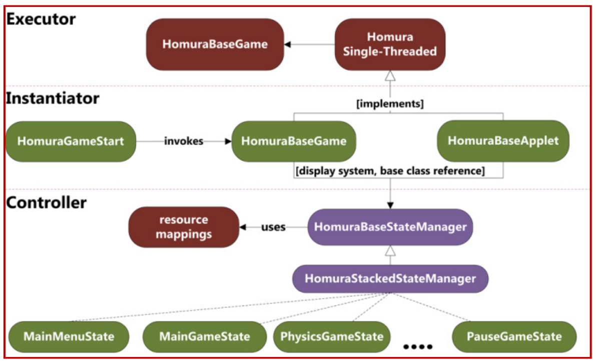


Figure 2: Game Partitioning

Executor: Its role is to define the game's update/render loop and application flow, independent of the application type. The executor handles hardware and general Java exceptions, incorporates the logging system, polls the input devices and generates input events as an event queue. The Executor also defines an interface with key methods (initialise, pre-update, update, post-update, render, cleanup) which each game must implement.

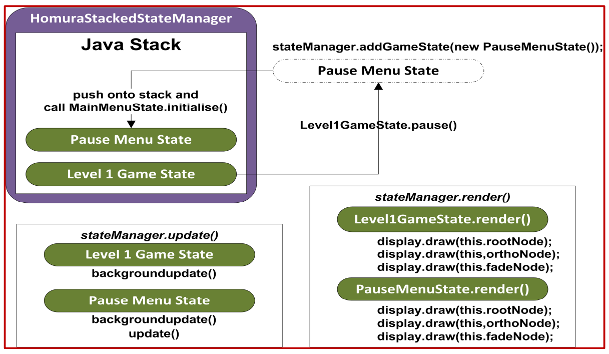
Instantiator: Its role is to create and configure the display system and renderer, assemble Homura's asset management system and concretely implement the *Executor* interface. The *Instantiator* creates an instance of a concrete *Controller* implementation and binds it to each of the key methods of the *Executor's* interface. Each application variant has a concrete version of an *Instantiator* to create the specific graphics context.

Controller: The backbone of the application. It provides the access point to the Display System, irrespective of application type, and provides contextual information regarding the current execution environment (graphics card capability, memory usage, OS version, screen resolution, colour depth, anti-aliasing etc) and provides core helper operations for the rendering system (create a camera, change the view frustum, create picking rays, convert between world and screen co-ordinates etc.). All game components utilise the controller in order to access the renderer and viewport, meaning the game is abstracted from its rendering target.

3.1.3 Game State System

Homura's game state system [14] allows the game to be logically organised into individual game screens, based on their functionality or purpose within the game system. This allows transitions between different sections of the game (e.g. GUI - loading screen - level 1) to be handled in a simplified manner, as well as encapsulating the game as logical sub-sections. Homura provides an abstract base class, *HomuraBaseGameState,* which all other game states inherit from. Each game state maintains three separate scene graphs - one for 3D objects (the Root Node), one for 2D objects (the Orthographic Node) and one for transitional effects (the Fade Node). This allows the developer to easily overlay 2D graphics on top of a 3D scene (e.g. HUD elements). This separation also limits the number of state switches OpenGL has to perform (e.g. transferring between perspective and orthographic projection) during the rendering phase. The base class provides a core set of pre-initialised objects: a camera viewpoint from which the 3D scene is rendered, a base lighting system to illuminate the scene and a Z-Buffer to sort the 3D objects from the viewpoint. The base class also provides several abstract methods to implement, each correspond to a key part of the game loop: `initialise()`, where assets should be loaded, objects set their base state, and the initial scenegraph constructed;

`backgroundupdate()` and `update()`,where game logic, AI, physics should be updated and any changes to the scenegraph should occur; and finally `render()`,where any additional scenes to the default should be passed for rendering. The base class automatically updates and renders the geometric and material states of the three provide scenegraphs. Individual game states are developed through extension of this base class. Homura provides several pre-implemented sub-classes, designed for commonly required game screen functionality such as *HomuraPhysicsGameState*, which adds the Physics node support, with standard gravity and friction setup; *HomuraBaseMenuState* which adds an extensible 2D/3D Menu system; and *HomuraDebugGameState* which adds runtime debugging support.



HomuraStackedStateManager
Java Stack
push onto stack and call MainMenuState.initialise()
Pause Menu State
Level 1 Game State
stateManager.addGameState(new PauseMenuState());
Pause Menu State
Level1GameState.pause()
stateManager.update()
Level 1 Game State
backgroundupdate()
Pause Menu State
backgroundupdate()
update()
stateManager.render()
Level1GameState.render()
display.draw(this.rootNode);
display.draw(this,orthoNode);
display.draw(this.fadeNode);
PauseMenuState.render()
display.draw(this.rootNode);
display.draw(this,orthoNode);
display.draw(this.fadeNode);

Figure 3: Homura State Management System

Figure 3 illustrates a commonly utilised state manager scenario. Game states are added to a Homura game by pushing a new instance of a game state onto the stack, this also binds the state manager to the game state, for bi-directional access. The manager's update loop iterates over all the game states in the stack from bottom to top. All game states have their `backgroundupdate()` method called, but only the topmost state has its `update()` method called as it is in focus. The manager's render loop also iterates over each of the game states in the same order as the update loop calling their `render()` method. This guarantees the order of rendering so that the 3D root node is rendered first, then the HUD node, then the fade node. This means that 3D objects placed in a state higher in the stack are drawn after the 2D object of its previous state, allowing layering. This game state system allows some typical game tasks to be carried out with ease. An example to illustrate the reduction in complexity afforded to the developer is an in game pause menu system, as seen in Figure 3. In this scenario, an existing game state called *Level 1* has an event handler triggered (e.g. pressing the 'p' key) which has called a method called pause(), This method creates a new instance of the *PauseMenuGameState* and adds it to the state manager. The state manager pushes this onto the stack and initialises this game state's scenegraph (comprised of menu items such as 'resume' or 'exit game'). This pauses the game instantly as *Level 1* is no longer the topmost game state, which means its update() method is not being called, so no input events or scenegraph changes are being made to this game state. When the *PauseMenuGameState* is terminated by the player (e.g. presses a button to resume the game) this game state is popped from the state manager and its `cleanup()` method called to de-allocate unused objects. Subsequently, Level 1 becomes the topmost state again and its `update()` method is called again, resuming play.

3.2 Net Homura – The Web Framework

The Net Homura framework is an Application Programming Interface (API) which aims to facilitate the creation of web-based distribution platform for Homura-based games. This section describes the application architecture, core feature set and key information regarding the implementation of the core classes which comprise the API. In the future, Net Homura will also be further expanded to provide a networking middleware, to integrate MMOG support into the platform, as detailed in [15].

3.2.1 *Systems Architecture*

Net Homura provides a PHP-based web API for the development of websites and web applications. Net Homura uses Object-Oriented PHP5 and is structured so that it will integrate into the common web application stack, which features an OS, web server, relational database (RDBMS) and server-side application language interface. Typical configurations supported by Net Homura are LAMP (Linux, Apache, MySQL, and PHP), WAMP (Windows, Apache, MySQL, and PHP) and WIMP (Windows, IIS, MySQL, and PHP). Figure 4 illustrates the typical architecture of a Net Homura based web application.

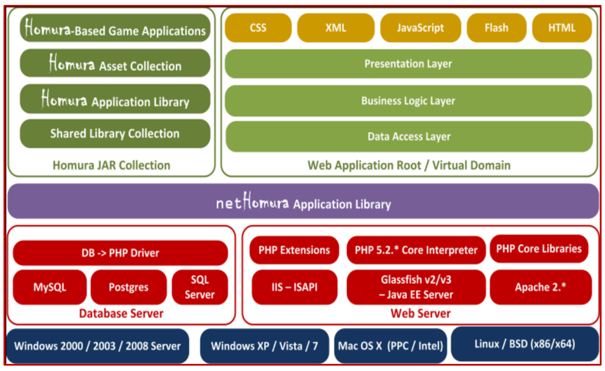


Figure 4: Net Homura Architecture

Net Homura is platform-agnostic, running on any OS which can run a PHP 5-enabled web server such as Apache or IIS. A typical Net Homura Application will comprise a data-access layer which maps data sources (e.g. a database) to the business-logic layer, which is comprised of PHP Objects, one for each system entities. The Presentation Layer is built using the Net Homura template system. A base application template is constructed by defining a sub-class of the Net Homura *Page* class. This template defines the base structure of the application (navigation, title etc); includes external scripts such as CSS and JavaScript; configures directory mappings to external content (Flash, videos, PDF, JAR files etc) and defines getters and setters for regions where dynamic content can be injected into the page.

3.2.2 *Deployment Overview*

Net Homura deploys the Java-based Homura games to the client machine in both JWS and Applet format. Net Homura provides a method to dynamically create JNLP configuration files and install games on the Client computer, as shown in Figure 5.

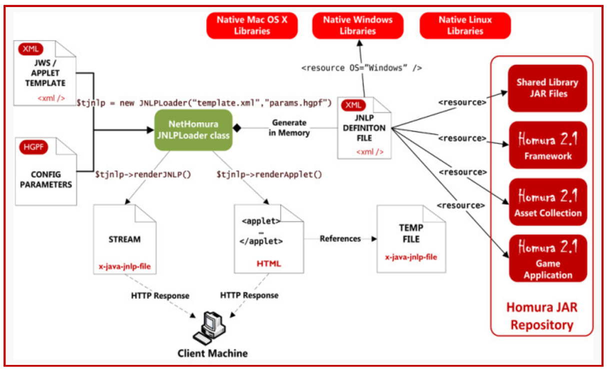


Figure 5: Deployment Overview

Deployment uses Net Homura's *JNLPLoader* class. An instance is constructed which takes two files as its parameters. The first is an XML template which contains the skeleton of a JNLP file. Customisable options are defined within this template as replaceable fields. Typical items which are configured are skinning information, security options, Java Virtual Machine versions and the Java JAR files which comprise the Homura game application. There are four types of JAR file *game, library, optional* and *native*: Game JARs contain the binary version of the main game application. The Library JAR files contain the Homura Asset Collection, Homura Framework and its library dependencies (see Figure 1). Optional Libraries are additional frameworks used by some of the games whilst the Native JARs contain the platform-specific libraries. The template automatically includes all the *Library* entities and defines the *Native* platform. The second file is a game-specific parameters file containing key-value pairs which specify the values of the template fields to replace. This must specify the game's application entry point based on deployment type. The *JNLPLoader* class parses the two files and performs template replacement to generate the JNLP file for supported application types. In order to create a JWS, `renderJNLP()` must be invoked sending the in-memory JNLP file as an HTTP Response stream, which is interpreted by the Client's Java Plugin and downloads the game. `renderApplet()` is invoked in order to create an Applet. This writes the in-memory JNLP to a temporary file and generates a HTML Applet tag, which references this temporary JNLP. This can then be embedded inside an HTML page and returned to the client which is interpreted in a manner analogous the JWS. The separation of game and libraries can drastically reduce bandwidth usage. On first usage, the player downloads the game and all the required Homura libraries, along with the natives for the Client platform. If the user downloads another game, only the main game JAR and the optional JARs need to be downloaded. Once downloaded onto the Client computer, all JARs are cached, so that any subsequent executions are executed from local storage. Additionally, since the Applet and Web Start use the same game JAR, once a JWS version has been downloaded, the applet version is executed automatically from the cache, and vice-versa.

3.3 Prototyping and Release

During the course of developing the Homura and Net Homura frameworks, we devised a series of seven test games and eight technical game demos which demonstrate all aspects of the Homura framework. Net Homura was used to construct a portal application [16], as described in [15]. This houses all the technical demos and successfully deploys the games applications in a cross-platform, cross-browser manner, whilst utilising all the features of the Net Homura framework. Both the frameworks have been released under LGPL licenses and are available from the official project site [6].

4. EVALUATION OF SOLUTION

This section provides an analysis of the platform's efficacy and suitability as a means for the deployment of hardware accelerated 3D games in a cross-platform, cross-browser manner. The Net Homura based prototype portal application and the nine Homura-based showcase demos, published in a real-world server setting, provided an appropriate test-bed for benchmarking and evaluating the Homura solution. These sample applications also demonstrate the technical viability of using JWS and Applets as a means for the deployment of advanced Java-based games applications. Figure 6 illustrates an example Homura application deployed directly from Internet Explorer in Applet format.

As well as satisfying the technical viability of this approach, the platform must demonstrate real-world applicability. The evaluation of is based on the criterion outlined in section 2.3 of this document. A more detailed appraisal of the system can be found in [15]:

- **Security:** The deployment platform web application can be secured via standard web security techniques. Net Homura deployment support HTTPS transfer. Homura games in both Applet and JWS form must be signed with RSA digital certificate, in order for the Java security system to allow access to native libraries, which can be used to identify the authenticity of the distributed software.
- **Integration with existing web technologies:** Net Homura features programmatic integration support for any PHP-enabled web server. Homura-based games can be embedded into any HTML-compatible framework using standard next-generation applet / JWS methods. The Net Homura application framework is extremely compact with a distribution size of 484KB.
- **Cross Platform Consistency:** Homura games are cross-platform compliant across the range of desktop PC platforms, supporting Windows 2000/XP/Vista/7 (x86/x64), Linux (x86/x64) and Mac OS X (PPC/Intel-based).
- **Cross Browser Consistency:** The Net Homura platform and Homura-based games applications are interoperable with the most common browsers available on a given platform (i.e. IE 6/7/8, Firefox 3, Google Chrome, Safari etc)

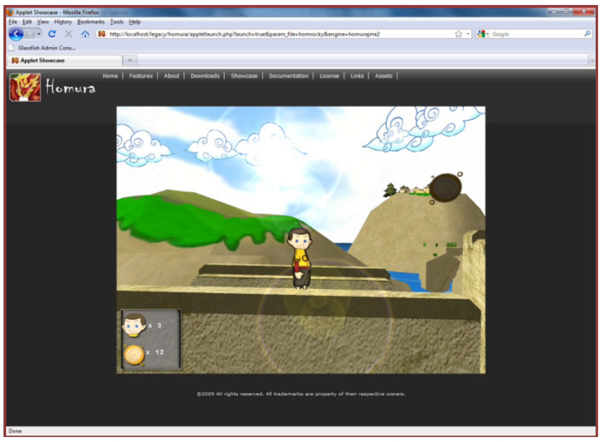


Figure 6: Homura Applet running on IE 8

4.1 Homura Application Performance

Homura's runtime performance has been previously evaluated in [17] using version 1 of the API. At the time of writing, version 2.1 has been released publicly and features some major improvements.

Table 1: Homura Performance Benchmarks

HARDWARE 1:	INTEL CORE 2 DUO E6600, 4GB DDR2-800 RAM, 256MB NVIDIA 8600GT.			
HARDWARE 2:	PENTIUM 4 3.6 GHZ, 2GB DDR2-666 RAM, 128MB ATI RADEON X1300.			
HW	OS	DISPLAY SYSTEM CONFIG	APPLET FPS	JWS FPS
1	A	800x600 Windowed 0xAA 24bit	192	210
		1024x768 Windowed 2xAA 24bit	176	190
		1024x768 Fullscreen 4xAA 32bit	N/A	193
1	B	800x600 Windowed 0xAA 24bit	211	254
		1024x768 Windowed 2xAA 24bit	200	217
		1024x768 Fullscreen 4xAA 32bit	N/A	222
1	C	800x600 Windowed 0xAA 24bit	190	211
		1024x768 Windowed 2xAA 24bit	176	201
		1024x768 Fullscreen 4xAA 32bit	N/A	192
2	A	800x600 Windowed 0xAA 24bit	163	170
		1024x768 Windowed 2xAA 24bit	153	164
		1024x768 Fullscreen 4xAA 32bit	N/A	168
2	B	800x600 Windowed 0xAA 24bit	169	178
		1024x768 Windowed 2xAA 24bit	155	172
		1024x768 Fullscreen 4xAA 32bit	N/A	175
2	C	800x600 Windowed 0xAA 24bit	160	169
		1024x768 Windowed 2xAA 24bit	148	165
		1024x768 Fullscreen 4xAA 32bit	N/A	153
OS A:	WINDOWS XP PROFESSIONAL SP3			
OS B:	WINDOWS 7 PROFESSIONAL			
OS C:	UBUNTU 10.04 X86 – LUCID LYNX			

The improvements focus on low-level API calls, VM optimisation and render queues in order to further increase its efficiency. Therefore, we subjected this version to the same series of tests as those detailed in [17] to assess our current levels of application performance. The procedural island demo from the

Homura technical demos was chosen due to its complexity. The scene is comprised of a dynamically generated height-map terrain system with multi-texturing and shadow mapping. This is then surrounded by a GPU-based water effect system. The entire scene comprises 262172 polygons. As before, the benchmarks were averaged over ten executions, with the frame-rate in Frames Per Second (FPS) averaged over a two minute execution time, after initialisation. The tests utilised Homura's in-built logging system, to ensure that the tests were as unobtrusive as possible. Table 1 details the results of the benchmarking. The tests were performed on two separate triple-booting machines, across varying resolutions and anti-aliasing configurations.

From these results, we can see that the optimisations have significantly improved performance since version 1. The results are expressed as *A-B (C)* where *A=minimum increase*, *B=maximum increase* and *C=mean increase*. Applets exhibited results of 48%-153% (94.5%) and JWS exhibited results of 43%-136% (87.3%). It must also be stated that these tests have also been influenced externally by JRE, OS and graphics card driver updates and OS types B and C have changed from Vista to Windows 7 and Ubuntu 9.04 to 10.04 respectively, introducing bias. However, OS A has remained unmodified at OS and driver level with only a JRE update, and exhibited results of 54%-135% (93.1%) and 46%-119% (82.7%) for Applets and JWS respectively, indicating that Java and Homura upgrades combined have considerably improved runtime performance. The trends between the test combinations have remained largely unchanged with resolution size and anti-aliasing proving inversely proportional to FPS, as expected. JWS variants still outperform their Applet counterparts exhibiting 7%-43% (17.3%). On the Windows machines, full screen mode outperformed windowed mode at similar display settings. Windows 7 was the fastest, XP second and Ubuntu third, across all tests. Crucially, all tests remained appreciably above the industry standard of 60 FPS.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented an overview of the Homura games framework and Net Homura deployment middleware and illustrated that Homura provides an integrated solution for Java based 3D games development. In section 5 we have presented the performance statistics for the games framework and highlighted the viability of Java as a language for games programming. We have also discussed the project's novel system architecture and prototype applications: which aim to unify the deployment of hardware-accelerated Java-based 3D games applications with online capabilities. Net Homura provides a multi-tiered deployment platform that is secure, robust, and easily portable to a wide range of web servers. The open source nature of the technologies used within the Homura and Net Homura framework allow the possibilities for the creation of a game which, from development through to hosting, deployment and networking, can be created with little or no financial outlay for the developer. This would enable small-scale developers to distribute modern games applications to users worldwide. Our future work will be geared towards further enhancements to the games framework, both feature-wise and performance related with particular focus being given to Java 7 support, data structure optimisations and concurrency (multi-threading, multi-core support).

The Net Homura framework is currently under extension to incorporate Java Enterprise Edition (EE) web technologies to provide an entirely Java-based solution, which will provide richer interoperability between the two systems. Both the Homura and Net Homura frameworks are currently being used to develop a test bed for research into scalability for P2P and Client/Server-based MMOGs, with TCP/UDP networking support being added to Net Homura. Homura is also being used for research into procedural content creation and facial animation, and is used as a teaching resource within the BSc and MSc Computer Games Technology / Animation courses run at LJMU.

REFERENCES

[1] (2010, June) Official Nintendo Wii European Site. [Online]. http://wii.nintendo-europe.com/334.html

[2] (2010, June) Official Sony Playstation Website. [Online]. http://uk.playstation.com/psn/store/

[3] (2010, June) Official XBox Site. [Online]. http://marketplace.xbox.com/en-GB/

[4] (2010, June) Official Steam Website. [Online]. http://store.steampowered.com/

[5] (2010, June) Official Apple App Store Website. [Online]. http://www.apple.com/iphone/iphone-3g-s/app-store.html

[6] (2010, June) Official Homura Project Website, LJMU. [Online]. http://java.cms.livjm.ac.uk/homura

[7] (2010, June) Unity Games Engine - Official Site. [Online]. http://unity3d.com/

[8] (2010, June) Official XNA Community Portal. [Online]. http://creators.xna.com/en-GB/

[9] (2010, June) Oracle - Java Developers Guide. [Online]. http://java.sun.com/javase/6/docs/technotes/guides/javaws/developersguide/contents.html

[10] (2010, June) Oracle - Technical Articles - New Applets. [Online]. http://java.sun.com/developer/technicalArticles/javase/newapplets/

[11] (2010, June) JNLP - JSR Specification. [Online]. http://jcp.org/en/jsr/detail?id=56

[12] (2010, June) EA Pogo Online Games Portal. [Online]. http://uk.pogo.com/home/home.do

[13] C Dennett et al., "3D Java Game Development with Homura," , Holiday Inn, Liverpool, UK, November 2008.

[14] C Carter, A El-Rhalibi, M Merabti, and M Price, "Networking Middleware and Online-Deployment Mechanisms for Java-based Games.," in *GDTW 2008, Sixth International Conference in Game Design and Technology*, Holiday Inn, Liverpool, UK, November 2008.

[15] C. Carter, "The Development of a Networking Middleware and Online Deployment Mechanism for Java-Based Games," LJMU , Masters Dissertation Jan 2009.

[16] (2010, June) Homura Games Deployment Portal Prototype. [Online]. http://java.cms.livjm.ac.uk/homuragames/login.php

[17] C. Carter, A. El-Rhalibi, M. Merabti, and M. Price, "Homura and Net-Homura: The Creation and Web-based Deployment of Cross-Platform 3D Games," in *Proceedings of Ubiquitous Multimedia Systems and Applications (UMSA)*, St Petersburg, 2009.