Homura and Net-Homura: The Creation and Web-based Deployment of Cross-Platform 3D 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@} ljm.ac.uk

Dr Marc Price
BBC Research and Development.
BBC Research
Tadworth, UK
Marc.Price@bbc.co.uk

Abstract— Digital distribution is becoming an increasingly important method within the games industry. The leading consoles each possess their own bespoke platform to digitally deploy games applications to their users via the Internet, whilst the Windows PC gaming market is catered for by systems such as Valve's Steam platform. However, these digital content services are often machine-specific, proprietary, utilising custom web frameworks and a rigid publication system. In this paper, we present Homura and Net-Homura; two interconnected frameworks which facilitate the development and deployment of cross-platform, hardware-accelerated 3D games applications using standard web browsers and web technologies, using a combination of Java and PHP.

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

I. 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 allow the users to easily access 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), 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 games 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.

II. 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.

A. 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. It requires a different plug-in for each browser supported (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. 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 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. There is currently no distribution platform for PC versions of XNA games, but games can be distributed on the Xbox 360 using the Microsoft community games portal of the Xbox Live Marketplace. 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].

B. 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 code to be run on different platforms, its safety features and its support, through the class loader system, for dynamically incorporating new code makes it particularly suitable for systems whose behaviour and configuration are expected to change over time. Nevertheless, until recently there was little exploitation of these qualities beyond the relatively trivial use of “applets” to spice up web page content. This is due partly to the relatively short time since the language was introduced in a stable form and partly to the lack of infrastructure components to support more complex applications distribution. This deficiency is being addressed by technology vendors who are developing architectures for systems distribution and deployment, based on or incorporating Java, and providing components within those architectures. 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 2, 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 to the original Java applet technology and have been modified to have architectural similarities to 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 simple, standardised XML schema to define several key aspects of the deployment process, such as skinning the deployment transfer window present to the client, the libraries which comprise the application, Operating System and architecture specific libraries, access and security permissions. There are sections within the schema which also define properties specific to either Web Start or Applets, such as the Web Start application entry point and the Applet class to invoke. Java Plugin 2 applications can be easily integrated into HTML pages as link to the JNLP description file (in the case of JWS) or as an <applet> tag with a reference to JNLP file. When the JNLP files are executed by a supported browser the JRE Plugin is invoked and subsequently handles the execution of the application. As a result, Plugin 2 based Java applications exhibit excellent cross platform, cross-browser support. The JNLP protocol also allows for greater freedom in JVM choice, offering the ability to provide automatic upgrades to the user’s Java Run-time Environment (JRE), whilst supporting side-by-side installation of multiple JVM versions. With the considerable improvements made by Java Plugin 2 to applets, 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, Pogo [12].

C. 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.
- **Open-Source:** The platform should be open in order to support a community dedicated to improving both the games engine and deployment platform.

III. System Overview

The Homura project [6] is comprised of three distinct sub-projects, designed to interoperate with each other 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, detailed in section 3A; Net Homura, a web-based Framework for the development of websites / web application, building pages in a scene-graph like fashion using Object-Oriented PHP / HTML / CSS / JavaScript. It easily allows the integration of Homura games into web pages, detailed in section 3B; 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 features a full Java programming editor framework, and a combination of visual editors to graphically construct and position objects within the game. The IDE project is outside the scope of this paper, and will not be covered in further detail.

A. Homura – The Games Framework

The Homura framework is an Application Programming Interface (API) which aims to provide an open-source platform to make it easy to develop 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.

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.

![Figure 1: Homura Application Architecture](image)

The bottom layer of the architectural stack is the System layer. Homura is a cross-platform framework and will run on Windows, Linux, Mac OS X, but requires an OpenGL compatible graphics card. The second layer is the Native Layer. Homura is written in Java but utilises native platform-specific libraries for the key sub-systems as this provides 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/Orbis for open source audio 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 the Java Monkey Engine (jME) to provide rendering and input handling functionality. Programmed entirely in Java, 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 provides a high performance scene-graph based graphics API. The scene-graph allows the organization of 3D geometry into a tree-like structure, where a parent node can contain any number of children nodes, but a child node must contain only a single parent. The nodes are organized spatially so that whole branches of the graph can be culled. This allows for complex scenes to be rendered quickly, as typically, most of the scene is not visible at any one time. The scene-graph’s leaf nodes consist of the geometry that will be rendered to the display. 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. Homura also integrates jME’s 3D Audio support. The audio sub-system again relies on LWJGL to provide the native bridge to the OpenAL audio library, whilst using the open-source OGG/Orbis system as the media format for audio files. Homura also utilises a jME sub-project, jME Physics 2 to provide the Physics simulation functionality of the framework. jME Physics integrates tightly with the jME scene-graph by virtue of its Physics object classes inheriting from the jME scene-graph Node class. jME Physics 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. The final library utilised by Homura is the integration of 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 scene-graph node called a JOPSNODE to allow them to be easily added into a scene, or attached to a game entity node (e.g. the exhaust of a car). 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
3.1.4) to provide the skeleton game - complete with all the aforementioned sub-systems. These classes are then implemented with the required game logic and the user-developed content (Models, textures, particle effects, music, sound effects, backgrounds, etc) which are stored as a Homura asset collection and loaded within the classes 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 such as Scala, Jython, JRuby and JavaScript (or any JSR223 compatible Scripting engine), scripts can easily be written to control any portion of the scene-graph (the whole scene to a single node) and can be used for a variety of purposes such as AI, cinematics, animation control, event triggers etc.

2) Features of the Framework

Homura provides an extensive set of features required of a modern games engine and utilises the benefits of Java such as its reflection system, platform-independence and large library of base packages. Some of the key features of Homura are:

- **Platform and Application Agnostic** – Games developed using Homura will run as Applets, Java Web Starts or standalone Java applications and will run on Windows, Mac OS X and Linux distributions that meet the minimum system specifications without requiring code modifications.
- **Platform Introspection** – Allows system-level properties to be queried to determine whether client can run a particular feature or perform a particular operation (GLSL Shader support, OpenGL 2/3 extension support, Anti-Aliasing and Anisotropic support, Nvidia/ATI extensions, memory usage, free space, OS information etc.)
- **Integrated Run-Time Debugging System** – Comprehensive debugging system with real-time statistical reporting (frame rate, vertex and poly counts), visual aids (normals, tangents, bounding boxes, physics forces, physics bounds, wireframe node etc), run-time, reflection-based scene graph introspection allowing dynamic traversal and modification of the graph node, Reflection-based console system to execute scripts, run commands, alter Java objects etc.
- **Multi-Format Asset Support** – Model loaders for key formats (COLLADA, 3DS, OBJ, MD5 etc) and Texture / Image loaders for main image formats (DDS,TGA,PNG,JPEG,GIF etc) with sprite-based and 3D animation support.
- **Game State System** – State system for the implementation of game-screens (GUI, 2D / 3D Scenes, Pause Menus, Loading screens etc). Handles garbage collection and object de-allocation for efficient memory usage.
- **Effects System** - GLSL Shader Support and Particle System support (JOFS/jME); integrated effect systems for water, texture-splatting, depth of field, bloom, cartoon-shading, Normal Mapping, Parallax Mapping etc.
- **Physics and Collision System** – Physics system which handles Rigid-body dynamics, ragdolls and material-based interactions. Physics and non-physics based collision systems and support for a variety of bounding geometries (AABB/OBB etc).
- **Game-Specific Optimisations and Common Techniques Library** – Fast Math approximations, Level of Detail, Lighting, Terrain Paging, 3D Audio support, Environment Mapping.
- **Scripting Support** – Supports JSR-223 compatible scripting languages allowing control over user-defined entities or scene graph objects.

3) Application Partitioning

The previous sections 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](image)

**Executor:** The role of the Executor 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. The Executor has a start() method which is used to start the game execution loop.

**Instantiator:** The role of the Instantiator 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 the Controller and binds it to each of the key methods of the Executor’s interface. Each application type has a concrete version of an Instantiator which creates the application-specific graphics context (e.g. HomuraBaseApplet creates an AWT canvas to embed inside a webpage, whilst HomuraBaseGame creates either a windowed or full-screen application).

**Controller:** The Controller is 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 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. The base controller HomuraBaseStateManager allows developers to build their game state system, or utilise Homura’s own stack-based state system to separate the game, and control transitions between game modes.

4) 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 to Loading to Level 1) to be handled in an easy manner, and partitions the game into easier to handle sub-
Homura provides an abstract base class, *HomuraBaseGameState*, which all game states inherit from. This provides the developer with a blank state to build on top of. 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 changes OpenGL has to perform, switching 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 from, 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; `render()`, where any additional graphs to the default should be passed for rendering. The base class automatically updates and renders the aforementioned main scenegraphs. Game screens 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 as mentioned in section 3A. This class can then be substituted with the base class when building the final product to remove debug support, making production builds a simple process. Game states are managed and controlled by a concrete implementation of the *HomuraBaseStateManager*. Game states utilise the manager to access the display system, get timing information and access the functionality provided by the Controller role. Homura provides *HomuraStackedStateManager*, which stores the game states in a stack. Figure 3 illustrates some state manager scenarios.

![Figure 3: Homura State Management System](image)

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. 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. Timing conditions can be set on the game states to specify a fade in and fade out duration when they are pushed on / popped off the stack so that the game state can query its current transition state as a value between 0-1 (where 1 is on top and 0 is overlaid). This can be used to apply transition effects such as a colour fade, transparency fades, slides etc. 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 illustrated 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 it’s `update()` method is called again, resuming play. This is all handled via the data structures and state system, requiring no coded logic in the game. The defined rendering order also allows easy visual effects to be applied to the pause system, such as adding a transparency to the *Level 1* state’s fade node so that the Pause Menu state’s text becomes more legible, when it is rendered on top of *Level 1*’s scene.

### B. NetHomura – The Web Framework

The NetHomura 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].

#### 1) Systems Architecture

Net Homura provides a PHP-based web API for the development of websites and web applications. NetHomura 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 below, illustrates the typical architecture of a NetHomura based web application. NetHomura is platform-agnostic, running on any Operating System which can run a PHP 5-enabled web server such as Apache 2 or Internet Information Services (IIS) Server. NetHomura also supports both open-source and proprietary database technologies such as MySQL, Postgres and SQL Server, as long as they have a supported PHP driver. The NetHomura PHP library can be included either server-wide or application-specific within the root of the web application directory / virtual domain.

- **Database Integration**: Abstracts the connection so that all access and usage with NetHomura is done without requiring specific knowledge of the RDBMS used.
- **Client Side / Server Side Validation**: JavaScript and PHP functions to validate user input at the browser and the server.

- **Base Data Type Support Functions**: These functions provide commonly needed routines for each of the main data types, such as Array Sorting, HTML to string conversion, password generation, hashing functions etc.
- **XHTML Page-Template class**: Encapsulates an HTML/XHTML document as an Object-Oriented construct, allowing the programmatic definition of the presentation layer (CSS, JavaScript, Meta Tags etc.). This class is abstract and is designed to be utilised as the basis to construct a site template through class-inheritance.
- **Tag Generation Functions**: These functions provide single line helper methods for producing common HTML tags and form data as PHP Strings.
- **Browser/User-Agent Detection**: These scripts allow the developer to query information such as the browser used, operating system etc.
- **XML Driven JNLP Support**: System to drive the dynamic creation of a JNLP definition file, which controls how Java downloads your application, as detailed in the next section.

3) **Deployment Overview**

Net Homura deploys the Java-based Homura games to the client machine in both Java Web Start 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](image)

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 template 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 JAR files for additional frameworks used by some of the games and the Native JARs contain the platform-specific native libraries. The Template automatically includes all the Library JARs and defines the Native JAR files based on the platform. The second file used is a game-specific parameters file containing key-value pairs which specify the value of the template fields to replace. This file must specify the entry point to the Homura game of either the Web Start, Applet or both. The JNLPLoader class then parses both these files and performs template replacement to construct the JNLP file for supported
application types. To create a Web Start, \texttt{renderJNLP()} must be invoked, sending the in-memory JNLP file as an HTTP Response stream, which is interpreted by the Client’s Java Plugin, which downloads and installs the game. To create an Applet, \texttt{renderApplet()} is invoked. 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 and the Java Plugin interprets this in the same way as the Web Start. 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 of an already played game use the cache, so the game runs instantly. Also, because the Applet and Web Start use the same game JAR, once a Web Start version has been downloads, the applet version is executed automatically from the cache, and vice-versa.

C. Prototyping and Release

During the course of developing the Homura and Net-Homura frameworks, we devised a series of six test games and three technical game demos which illustrate 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: \url{http://java.cms.livjm.ac.uk/homura/}, along with the release documentation. Live versions of all the demo applications are also available under the ‘Showcase’ section of the site, in both JWS and Applet format. A live sandbox version of the Homura Games web applications portal, built using Net-Homura, is available for public preview at: \url{http://java.cms.livjm.ac.uk/homuragames}.

IV. SOLUTION ANALYSIS

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 prototype NetHomura-based 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 Web Starts and next-generation 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.

A. Evaluation of Solution

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 2C of this document. A more detailed appraisal of the system can be found in [17]:

- **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 web start 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 / web start methods.[10] 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 (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 following browsers in a standards compliant, consistent manner:

<table>
<thead>
<tr>
<th>Deployment Method</th>
<th>IE6</th>
<th>IE 7</th>
<th>IE 8</th>
<th>FIREFOX 2.0+</th>
<th>FIREFOX 3.0+</th>
<th>CHROME 1.0+</th>
<th>SAFARI 3.0+</th>
<th>OPERA 9.0+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applet</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Web Start</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **Download Size:** The Homura framework download size is compact, requiring 14.1MB for the entire Homura Framework, Homura Asset Collection and additional APIs. The JAR files are compressed using gzip and Homura also supports Pack 200 compression.

- **Application Updates:** Both Web Start and Applets support application updating through the JNLP specification. This can be achieved declaratively using Net-Homura, or programmatically using Homura in combination with the JNLP API (\url{java/jnlp} package).

- **Open-Source:** Both Homura and Net-Homura have been released under an LGPL license. This license is less-prohibitive in that commercial or closed source titles can be built on top of the framework, but any additions, fixes, improvements made to the underlying platform must be committed back to the community, in order to help the platform progress.

B. Homura Application Performance

Whilst the other aspects of the evaluation are important to the overall quality of the solution, the main criteria for assessment must be the performance of the Homura games, within their deployed context. The procedural island demo from the Homura technical demos was chosen due to its complexity. The scene is comprised of a terrain dynamically generated from a height map and multi-textured with four
texture layers. The island is then surrounded by a GLSL shader implemented water effect system. The entire scene comprises 262172 polygons. The games application was run with the browser as the only active application context. The benchmarks were averaged over ten executions, with the frame-rate averaged over a two minute execution time, after initialisation. The Homura applet implementation is currently capped to update at 60 frames a second (a limitation currently imposed by LWJGL), but due to the architecture of the application, the timer object could be tested independent of this restriction. The tests utilised Homura’s in-built logging system, to ensure that the tests were as unobtrusive as possible. Table I details the results of the benchmarking. The tests were performed on two separate triple-booting machines.

<table>
<thead>
<tr>
<th>HARDWARE CONFIGURATION</th>
<th>DISPLAY MODE</th>
<th>APPLET FPS</th>
<th>JWS FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Windows XP SP3</td>
<td>800x600 Windowed 9xAA 24bit</td>
<td>124</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>1024x768 Windowed 2xAA 24bit</td>
<td>78</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>1024x768 Fullscreen 4xAA 32bit</td>
<td>N/A</td>
<td>105</td>
</tr>
<tr>
<td>2: Windows Vista SP2</td>
<td>800x600 Windowed 9xAA 24bit</td>
<td>124</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>1024x768 Windowed 2xAA 24bit</td>
<td>79</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>1024x768 Fullscreen 4xAA 32bit</td>
<td>N/A</td>
<td>106</td>
</tr>
<tr>
<td>1: Ubuntu 9.04 with vendor's Binary graphics driver support</td>
<td>800x600 Windowed 9xAA 24bit</td>
<td>128</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>1024x768 Windowed 2xAA 24bit</td>
<td>83</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>1024x768 Fullscreen 4xAA 32bit</td>
<td>N/A</td>
<td>99</td>
</tr>
<tr>
<td>2: Windows XP SP3</td>
<td>800x600 Windowed 9xAA 24bit</td>
<td>104</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>1024x768 Windowed 2xAA 24bit</td>
<td>65</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>1024x768 Fullscreen 4xAA 32bit</td>
<td>N/A</td>
<td>95</td>
</tr>
<tr>
<td>2: Windows Vista SP2</td>
<td>800x600 Windowed 9xAA 24bit</td>
<td>104</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>1024x768 Windowed 2xAA 24bit</td>
<td>64</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>1024x768 Fullscreen 4xAA 32bit</td>
<td>N/A</td>
<td>95</td>
</tr>
<tr>
<td>1: Ubuntu 9.04 with vendor's Binary graphics driver support</td>
<td>800x600 Windowed 9xAA 24bit</td>
<td>105</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>1024x768 Windowed 2xAA 24bit</td>
<td>67</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>1024x768 Fullscreen 4xAA 32bit</td>
<td>N/A</td>
<td>90</td>
</tr>
</tbody>
</table>

From these results, it is clear that performance suffers slightly when comparing Applets to Web Starts, with a performance penalty of around 8-10% expected. The difference in performance between Windows versions was negligible, with the Linux implementation slightly faster in windowed mode, but slightly slower in full-screen. As expected, increased resolution and anti-aliasing resulted in performance degradation, whilst full screen performance was markedly better than achieved by their windowed mode counterparts. The performance in each case also never dropped below 60FPS, the current performance standard expected from modern games applications.

V. CONCLUSION

In this paper we have presented an overview of the Homura Engine and Net-Homura deployment middleware. Homura provides an integrated solution for Java based 3D games development. We have also discussed 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. The networking middleware provided 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, and combined with the Homura engine and IDE, 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. This would enable small-scale developers to distribute modern games applications to users worldwide. There are several enhancements and future directions that can be taken in the future development phases in order to fully realize the potential of the proposed solution. Currently, the content of the system is directly added to the database via a set of scripts and stored procedures. The deployment system can be further developed to provide a set of back-end tools which allow the management of games, users, and application configuration through a web interface. This should be a relatively straightforward implementation, using the existing data-access and application tiers. The middleware component requires the completion of the test games in order to evaluate its performance and scalability properly. The first phase is to support between 8-16 concurrent players during a game session. The next phase will involve the incorporation of networking algorithms such as dead-reckoning, Area of Interest Management, cheat-detection to increase the scalability of the system by an order of magnitude.

VI. REFERENCES