1998

Bamboo - A Portable System for Dynamically Extensible, Real-Time, Networked, Virtual Environments

Watsen, Kent

Appears in the proceedings for the 1998 IEEE Virtual Reality Annual International Symposium (VRAIS'98), held March 14-18, 1998 in Atlanta, Georgia.

Bamboo -
A Portable System for Dynamically Extensible, Real-time, Networked, Virtual Environments

Kent Watsen and Mike Zyda
Naval Postgraduate School, Monterey, Ca.
watsen@acm.org, zyda@siggraph.org

Abstract

Bamboo is a portable system supporting real-time, networked, virtual environments. Unlike previous efforts, this design focuses on the ability for the system to dynamically configure itself without explicit user interaction, allowing applications to take on new functionality after execution. In particular, this framework facilitates the discovery of virtual environments on the network at runtime.

Fundamentally, Bamboo offers a compatible set of mechanisms needed for a wide variety of real-time, networked applications. Also included is a particular combination of these mechanisms supporting a dynamically extensible runtime environment.

This paper serves as a general introduction to Bamboo. It describes the system's architecture, implementation, and future directions. It also shows how the system can facilitate the rapid development of robust applications by promoting code reuse via community-wide exchange.

Keywords:

portable, multi-platform, dynamically-extensible, real-time, networking, virtual environment, virtual reality, toolkit, framework, system

1: Introduction

The development of virtual environment (VE) applications has been of academic and commercial interest ever since the advent of three dimensional (3D) graphics. These applications are typically written from scratch on top of the native system and low-level graphics libraries. Only recently have standard higher-level graphics libraries, such as OpenGL++ [25], providing cross-platform scene graph construction and manipulation become available. Even so, the process of developing robust applications still requires an immense amount of effort. Recognizing this trend, some academic and commercial institutions have developed toolkits facilitating the development of specific applications by providing features common to that type of application. However, these toolkits tend to be monolithic architectures, limited in capability, difficult to extend, and/or available on only a few platforms, if not just one. The most significant of such toolkits include Alice [8], AVIARY [27], BrickNet [26], DIVE [5], dVISE [9], EasyScene [6], MASSIVE [11], NPSNET [13], Vega[20], MR Toolkit [10], VEOS [3], and World Toolkit [24].

The architecture described in this paper, Bamboo, is the result of years of trying to develop the right toolkit for the research and development of networked VEs. Historical observation suggests that no single toolkit can hope to address all user needs, while modern trends seem to indicate that such goals may be achieved by facilitating the distribution of the effort to a community of users, such as with an extensible architecture like Bamboo.

Bamboo is an attempt to enable dynamically scalable virtual environments hosted on the network. It achieves this goal by understanding key issues and providing direct support for them, applying lessons learned from previous efforts towards an efficient implementation. These solutions are provided in the form of practical mechanisms implemented using object oriented and generic programming techniques. Furthermore, and perhaps more significantly, the general design philosophy has been to be minimally invasive, resulting in higher code reuse and lower learning curves, thus increasing productivity while reducing chance for error.

2: VE Toolkits, Frameworks, and Systems

Toolkits, in general, are designed to facilitate the development of applications by providing functionality that reduces the user's programming effort. A variety of functional characteristics may be present in a toolkit including common mechanisms, specific procedures, compatibility, portability, performance, and extensibility.

Toolkits are typically presented to the user in the form of an application programmer's interface (API) implemented by header files and linkable libraries. Some toolkits, known here as frameworks, enforce a structure on the execution of the program, and may or may not embed the "main" routine. Frameworks that do embed the "main" routine, known here as systems, may or may not allow modification to the executable after compilation. Systems, and applications in general, that do enable self-modification at runtime are said to be dynamically extensible. VE-specific toolkits, frameworks, and systems are simply those that facilitate the task of developing VE-specific applications.

It may be illustrative to now point out some graphics-specific toolkits, frameworks, and systems. A couple toolkits providing low-level graphics support include OpenGL [17] and Direct3D Immediate Mode [14]. Some frameworks providing mid-level graphics support include Performer [21], Direct3D Retained Mode [15], and Java3D [7]. Some systems providing high-level graphics support include Alice, EasyScene, and NPSNET. Note that each of these progressions tends to utilize the capabilities of the lower layers. However, each progression extends the lower layer in some way. For instance, Performer adds scene graph and App/Cull/Draw semantics to OpenGL, while EasyScene is a Performer-based extensible system adding inter-object relationships, among other capabilities.

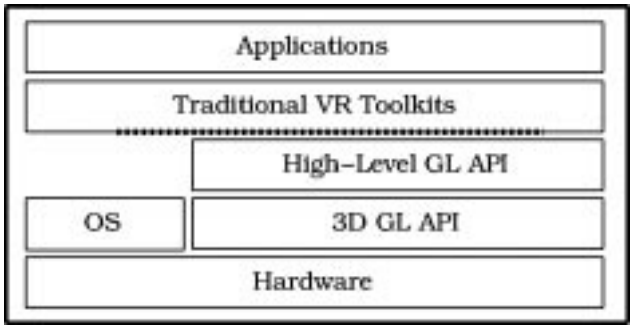

Figure 1: Traditional Toolkit Layering

However useful toolkits are meant to be, none are beyond criticism. Some are large in size (bloated), complex in nature, limited in capability, difficult to extend, and/or not available on multiple platforms. A particular distinction to be emphasized is whether a lower toolkit layer is completely encapsulated by a higher layer or if that access is left available and, if so, could its use inadvertently affect one or more higher layers. This last point, illustrated by the dotted line in Figure 1, is significant in that conflicting states between layers may lead to confusion to the user and/or disaster for the executable.

3: Initial Design and Implementation

Named after the exotic plant known to flourish, Bamboo is designed to facilitate the research and development of VE applications on multiple platforms while not being limited by its own implementation. It is the product of insight through years of experience developing VE applications and the need for support more flexible than provided by current offerings. Only through this synergism of need and insight is Bamboo able to be an effective solution while avoiding the usual pitfalls.

Bamboo is unique in many of its implementation and capability characteristics. The most notable design attribute is its provision for dynamic extensibility achieved through fully embracing the plug-in metaphor popularized by the commercial packages PhotoShop [1] and Navigator [18]. However, assuming large-scale networked environments and the need for explicit linking at runtime, Bamboo extends the original plug-in metaphor by adding inter-module dependencies and intra-module security, the purpose of which are detailed later.

A notable design requirement is for the system to be portable. This decision recognizes other systems, most notably the PC, as emerging cost-effective alternatives to the more traditional SGI hardware. Being portable can mean either that the source code can be compiled into multiple-platform-specific binaries or that the source code can be compiled into an intermediate form that can either be translated or turned into a platform-specific binary at load-time. Although the latter choice offers greater flexibility, there are not many solutions enabling it. Therefore, it was decided that Bamboo and all of its dependencies would have to be compiled for each target platform.

A likely implementation solution would have been to use Java [12], but the language is not yet considered suitable due to the real-time performance requirement immersive experiences demand[1]. The only other reasonable choice is C++, for its execution speed, strongly typed preprocessor, and support for object-oriented and generic programming semantics. However, unlike Java, the "standard" C++ libraries vary from system to system. This concern coupled with the fact that these libraries lack both type checking and object encapsulation suggested the use of some foundation class library. Several such toolkits include Washington University's ADAPTIVE Communication Environment (ACE) [23], ObjectSpace's Systems<Toolkit> [19], RogueWave's Total Solution Suite [22], and University of South Carolina's Yet Another Class Library (YACL) [28]. Furthermore, in

[1] This decision should be reconsidered in a few years as hardware becomes faster and the language matures.

addition to providing the portable standard libraries, including the standard template library (STL) [16], these toolkits also provide networking, concurrency, and synchronization abstractions, as well as offering CORBA [2] and Java interoperability. ACE is chosen among these for not only being a leader but also for being provided free of charge.

Although ACE offers many of the desired capabilities for a multi-platform, networked, virtual environment toolkit, it does not offer any graphics support. Fortunately, SGI has been developing a multi-platform, OpenGL-based, scene graph, C++ API to be known as OpenGL++[25]. From Bamboo's perspective, OpenGL++ offers the graphics, as well as window, keyboard, and mouse events.

Depending on how it is used, Bamboo is a toolkit, a framework, and a dynamically extensible system. It is a toolkit in that it exposes an API for a variety of mechanisms that may or may not be used. It is a framework in that a few of the mechanisms enforce a structure to its execution. And it is a system in that the runtime environment represents a particular combination of mechanisms with a "main" routine enabling dynamic extensibility. Bamboo's mechanisms and the runtime environment are detailed in following sections.

Bamboo's layering diagram is depicted in Figure 2. Note that although the Bamboo API is depicted as sitting on top ACE (due to its dependency), it can conceptually be thought of as sitting along side of it and OpenGL++. It should also be noted now that Bamboo's kernel is extremely small, having just enough logic to load in plug-ins. That is to say, it is completely up to the plug-ins to give the application its functionality.

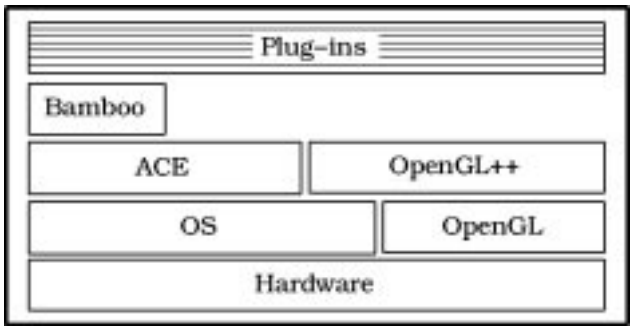

Figure 2: Bamboo's Layering

4: Bamboo's Mechanisms

Bamboo's mechanisms provide the core components needed to support the development of dynamically extensible applications. The main focus of these components is to enable the coexistence of plug-ins in a multi-threaded environment.

4.1: Object Database and Type Identification

This mechanism is implemented by inserting class-specific templated code into a class's definition. It provides simple routines for the identification, storage, retrieval, naming, reference counting, and thread-safe storage for objects of that class. Once specified, these routines automatically maintain themselves, negating the need for the application to provide similar functionality.

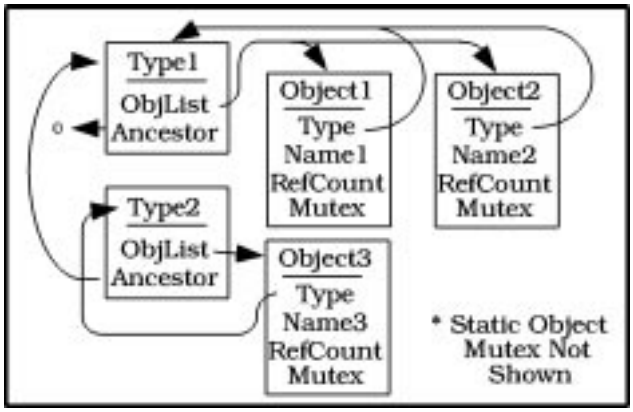

Figure 3: RTTI, Naming, Reference Counting, and Database Management

The structures needed in memory to support the database and runtime type identification (RTTI) routines are completely specified in each object's base class and an associated type class, depicted in Figure 3. These routines enforce the following relationships:

- class types reference their derived class types, thus providing ancestor relationships
- class types may optionally maintain a reference to instances of its class type, thus providing simple database queries
- every object references its class type via a meta-variable, thus providing for RTTI
- every object must have a unique name, if one is set, thus providing named lookups
- every object knows the number of times it is referenced, thus facilitating system maintenance
- every class maintains a static synchronization primitive, thus enabling thread-safe access to class-specific data
- every object maintains a synchronization primitive, thus enabling thread-safe access to object-specific data

4.2: Callbacks

The callback is one of the most fundamental mechanisms in all of Bamboo. In its simplest form,

depicted in Figure 4, a callback abstracts the execution of a single function having a specific declaration. In particular, the callback abstraction passes one reference to the invoking object and another to some user-specified callback data. As trivial as it may seem, the callback will be seen to be the backbone of Bamboo's architecture.

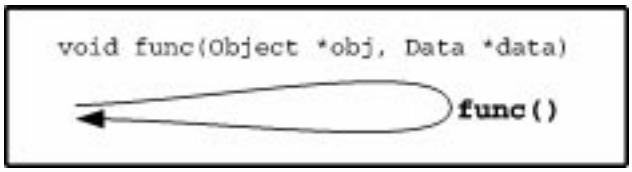

Figure 4: A Simple Callback

4.3: Extensibility

The ability to extend a system leads to solutions with unlimited capability. An extension can be thought of as an addition to an already existing structure. In the context of computer architectures, extensibility may refer to the ability to extend:

- a single object or a whole class
- the executable code and/or support structures
- the execution behavior of the program itself

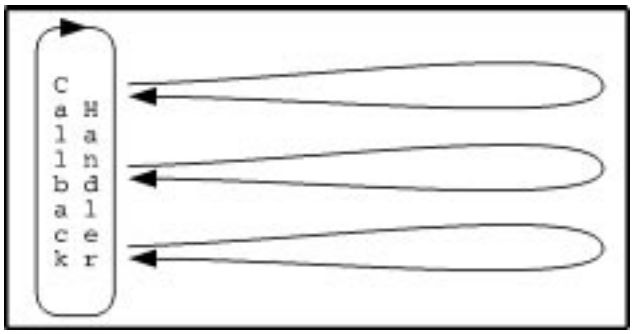

Figure 5: The Callback Handler

Bamboo directly supports all three of these forms of extensibility. In particular, because the C++ class declarations are made available, it is possible to inherit off them. Secondly, Bamboo has the built-in ability to dynamically load modules into the executable's memory space. Using dynamic linking instead of static linking offers several advantages. Dynamically linked libraries save memory, reduce swapping, save disk space, and upgrade easier. Finally, each module has an opportunity to attach itself to and remove itself from the process's execution loop when being paged in and out of memory. This last point is implemented in the form of callbacks being attached to a callback handler as depicted in Figure 5. Since callback handlers derive from objects, they can be named and therefore easily locatable.

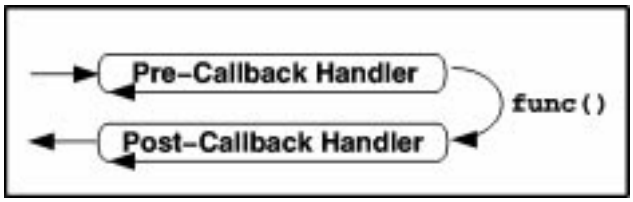

Figure 6: All Callbacks Are Recursive

Of course, the callback handler itself would be a limiting solution if left in this form. Ensuring that the system can support robust behavior, each callback is actually recursive in that it embeds two callback handlers (see Figure 6), one just before and one just after the callback function is executed. This approach facilitates the grouping of like functionality. For instance, rendering engines typically implement app, cull, and draw stages as a pipeline. Users are expected to place code in areas before and after each stage. These areas are usually referred to as pre-app, post-app, pre-cull, post-cull, pre-draw, and post-draw. In this way, the executable can be thought of as a tree of callbacks (see Figure 7). Any sub-tree of this execution tree may be selectively pruned or simply paused, automatically doing the same to its children.

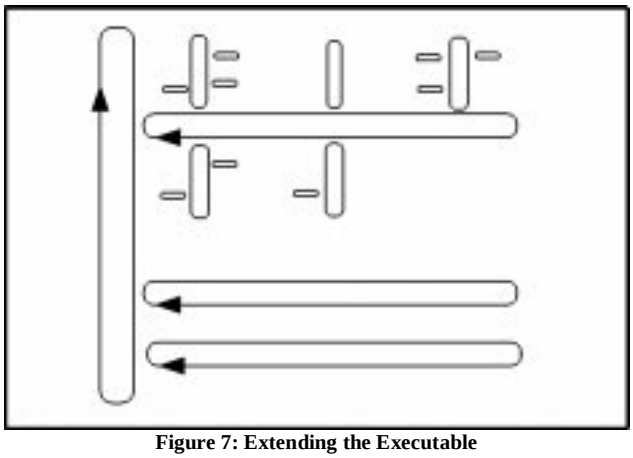

Figure 7: Extending the Executable

4.5: Event Handling

This mechanism provides an abstraction for the handling of system and user generated events. The event handler utilizes the callback handler in that notification of the occurrence of an event is given to registered parties via callbacks. Because a callback handler is used, multiple callbacks may be executed in response to a single event. Furthermore, each callback may have non-empty pre and/or post callback handlers.

Although both ACE and OpenGL++ offer event-handling mechanisms, neither appears to be general purpose enough for Bamboo. For instance, ACE's event

handler is best suited for system interrupts and demultiplexing of protocol keys, while OpenGL++'s constrains itself to just the window, mouse, and keyboard events. If either of these two mechanisms become more flexible, the event handler may no longer be needed.

4.6: Device Management

This base class abstracts the variety of devices that may be attached to the system. Some standard, and therefore supported, devices include the monitor, speaker, mouse, and keyboard. Some of these devices have input, some have output, and some have both. However, every device has an optimal cycle rate, the management of which leads to better resource utilization and system performance. Specifically, each device may be maintained in a its own light-weight thread that may be executed at a frequency suitable to that device.

Furthermore, this base class will (not yet implemented) attempt to provide common device data translation methods. For instance, a move forward callback may be executed in response to a specific keyboard, mouse, spaceball, joystick, or data glove event. Such methods may facilitate the introduction of new devices into existing applications.

4.7: Threading

Given that systems containing multiple processors offer significant benefits and are becoming more accessible, Bamboo includes this mechanism to facilitate the distribution and collaboration of multiple light-weight threads of execution. Threads are implemented in Bamboo using threaded callbacks, which are simply callbacks that are executed in a separate thread. All of the original callback semantics are valid. Particularly, the new thread executes the pre and post callback handlers.

Unpredictable events leading to a need for more CPU time than available may occur in even well behaved applications. Of particular interest is how the system's performance degrades under stress and if it can be managed gracefully. This mechanism will (not yet implemented) attempt to reduce the average processing load by symmetrically balancing each thread's CPU allocation until the stress has been removed.

4.8: Networking

The ability for applications to communicate over a network is a sensible, if not necessary, feature given current market trends. Therefore also central to Bamboo's architectural design are mechanisms supporting the current IP4 and forthcoming IP6 protocols. These abstractions address unicast, multicast, and broadcast packets for both reliable and unreliable transmissions.

Fortunately, it is not necessary to develop these interfaces as ACE already provides robust networking support. As its acronym implies, ACE has primarily been developed as a research tool for telecommunication industry. It provides very elegant abstractions for Internet addresses, sockets, streams, and datagrams. Furthermore, Bamboo will leverage off ACE's transition to IP6 and adoption of the reliable multicast protocol (RMP).

4.9: Graphical User Interface (GUI)

User interfaces are necessary in many applications, yet their efficient implementation within virtual environments is poorly understood. The confusion is in how to implement the GUI so that user interaction is not limited to the frame rate of the rendering engine. The solution is for the GUI to be in its own thread of execution.

Traditionally, this meant running the GUI in a separate process and having it communicate with the core process via remote procedure calls (RPC) over an inter-process communication (IPC) mechanism. However, the advent of lightweight threads enables a GUI to exist in the same process space and therefore execute core functions directly, provided that it does so in a thread-safe manner. Simple GUIs using this approach have been implemented for Bamboo with standard X-Windows and MS-Windows. However, neither of these windowing APIs are portable. Both Java's[2] abstract windowing toolkit (AWT) and TCL offer highly portable interfaces but also have limitations; AWT doesn't provide platform-specific look-and-feel, while TCL must reinterpret all of its scripts whenever a change occurs. The current implementation uses AWT for ease of use.

The other significant feature about the GUI is that it too must be dynamically extensible. Every time a module is dynamically linked into the core executable and the GUI is running (it may not be), the GUI support for the module must also be dynamically linked into the GUI. Figure 8 shows how the GUI might look before and after a module is brought into the system. Of particular significance, note that the GUI's menu bar initially displays the usual File, Edit, and Help items. To facilitate an intuitive interface, GUI panels being inserted as menu bar items must define the "path" from the menu bar to the panel. For instance, in this example, mod1 defines two sub-menu bars, both wishing to be placed under the "Objects" menu item. When Panel1 as being loaded, the

[2] Amazingly, the Java virtual machine (VM) can be executed from within a C thread

Objects item does not yet exist, so it creates it and then loads itself as a sub-item. Panel2 simply verifies that Objects exists and then loads itself as a sub-item.

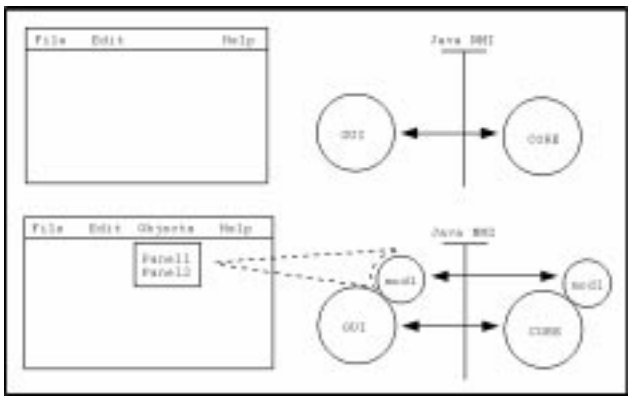

Figure 8: Java-based GUI Extensibility

Finally, it is recognized that there are VEs, such as those that use head-mounted displays (HMDs) or CAVEs, that have no use for the traditional 2D GUI interface. Applications that do not require an interface will note that the GUI itself is a module that can be paged in and out of an executable.

4.10: Physically-based Modeling

Too many virtual environments today are static in nature, having little if any motion. Movement is important because it results in more compelling scenes, which may lead to greater immersion. Many environments can be enhanced by physically-based logic that affects particular elements in the scene. However, it is not in the scope of this paper to define what logic is or is not needed for all applications using Bamboo. However, observation of systems that do implement physically-based models suggests that there exists two types of variables: global and local. Global variables typically specify environmental constants such as gravity, time of day, and wind direction. While local variables typically specify object-specific attributes such as mass, thermal conductivity, and elasticity. Even though global variables should be implemented as shared constants, some systems have each object locally define and maintain its own set of such variables. This situation potentially leads to some objects maintaining different global states. Therefore, this mechanism simply defines an environment object that enables global variables to be registered and then referenced by physically-based objects. The creation of an environment object is significant as it establishes a convention by which potentially diverse physically-based objects can share global states.

5: The Runtime Environment

As previously mentioned, not only does Bamboo provide a collection of commonly used mechanisms, but it also provides a particular combination of these mechanisms tied together with a “main” routine, forming a specific executable referred to as Bamboo’s runtime. Also mentioned is the ability to dynamically link modules, thus enabling for an executable to be dynamically extended at runtime. What was not emphasized, though, is that all of Bamboo itself is comprised of many modules, such that the original executable, the core kernel (see Figure 9), need only have enough logic to page modules and provide the initial framework for the plug-ins to hook into. In this way, no assumptions are made regarding what capabilities are needed by the kernel, but are determined at runtime by the application being loaded. For instance, if the particular application does not need protocol-specific networking support, the system would not load that module and thus save the memory and processing time that would ordinarily be consumed by such a mechanism.

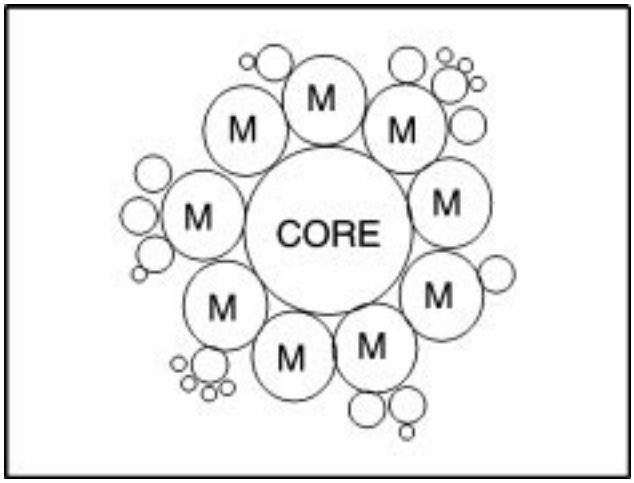

Figure 9: Runtime Abstract View

However, having each application specify every module it depends on could be a complex and error-prone process. Fortunately, each module, when being loaded, need only verify that its immediate dependencies are already in memory, loading them if not. For example, using Figure 10 as a reference, assume that M3 has already been loaded. In the process of trying to load M4, the system must first verify that M2 is in memory. M2 is not already in memory and must be loaded. In the process of trying to load M2, the system must first verify that M1 is in memory. M1 is already in memory, from when M3 was loaded, and does not need to be loaded again. Finally, M2 and then M4 may finish loading themselves.

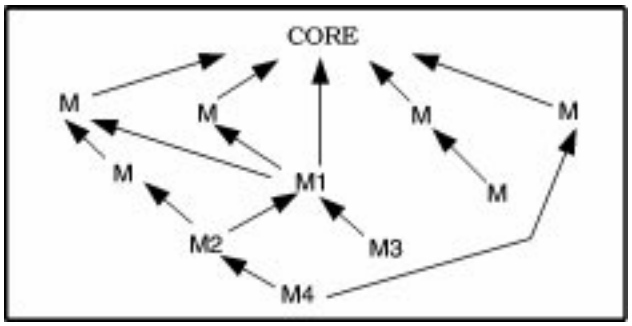

Figure 10: Module Dependency View

Because it is desirable to be able to have a module loaded off the network, if not found locally, its integrity may be suspect. This concern will be (not yet implemented) mitigated by Bamboo's insistence that a trusted partner sign all modules being loaded off the network. If the module does not have a trusted signature, the system prompts to have the signature added, to just load the module, or to ignore the module altogether. Modules may be multiply signed, thus enabling a hierarchy of trusted partners. For instance, the author may sign the module and the author's company may sign the module. Few may trust the author directly, but many may trust the author's company. At the top of this hierarchy is Bamboo itself; a module signed by Bamboo is implicitly trusted by all. It is recognized that this scheme can not be trusted to secure global-wide simulations, but does provide decent trust for academic and commercial institutions.

6: A Demonstration Scenario

A scenario might best illustrate how Bamboo may benefit an application. Imagine launching Bamboo by executing the runtime with just the GUI module (see Figure 8 for an illustration of the GUI's initial state). At this point the user might go to File->Load Module and select one of the initial modules in the distribution, a VRML [4] viewer for instance. This module might depend on the keyboard and a mobility model, both defined in separate modules. Furthermore, the mobility module may depend on the mouse for input. All of these modules are loaded into both the kernel and the GUI.

At this point, the user interacts with one of the newly defined GUI panels to load a VRML file and the user begins to fly through the world. A bounding box to some Bamboo node might be just inside the far clipping plane, however this node is not found locally and therefore must be downloaded from the network. A thread is spawned to asynchronously download the module, verify its signature, and unpackage it in its own "`shoots`" directory (see Section 7).

Assuming that this module defines a physically-based model of a flag, its directory might not only contain the geometry, textures, and sounds associated with the flag, but also the executable code defining its physically-based logic and a GUI panel enabling interaction with its local attributes (See Physically-Based Modeling above). Because the flag requires global variables such as gravity, wind direction, and wind magnitude to be defined, the environment module is also loaded into both the kernel and the GUI, thus enabling interaction with these variables as well. This example scenario will be a demonstration shipped with the final release distribution.

7: The Release Distribution

Extending the "bamboo" analogy, the release distribution is partitioned into two main sections: `roots` and `shoots`. The `roots` directory contains internally developed or officially adopted mechanisms, upon which a plethora of offshoots may be developed. The `shoots` directory, initially empty, contains external mechanisms and supporting structures (geometry, textures, sounds, etc) which may be downloaded during the course of a typical networked scenario. Both the `roots` and the `shoots` directories contain modules, each module represented by a single directory, the name of which defines the name of the module itself. A module's directory may contain the subdirectories `include`, `lib`, and `src` (if included by author); among others such as `geometry`, `sounds`, and `textures`. The purpose of this segmentation is for easy user identification and deletion of no longer needed modules, thus reclaiming potentially scarce storage space.

A few other root-level directories, `main` and `demos`, are also shipped with the release distribution. The `main` directory holds the default executable. As will be seen, source code available, the main routine is less then a hundred lines long – just enough code to initialize an execution loop (a callback handler) and read in command line arguments (modules) - loading them into the system. The `demos` directory holds a few example modules for using the mouse, keyboard, and graphics.

8: Cross-Platform Compatibility

Bamboo is portable to many platforms because it uses only standard APIs (C++, Java, STL, JGL, and OpenGL) and other multi-platform toolkits (OpenGL++ and ACE). Although this approach does not necessarily secure portability, the current system's concurrent development on several platforms has not been hindered thus far.

9: Availability

Bamboo is scheduled for a mid-1998 release, although beta versions are currently being made available. As previously suggested, the distribution is a collection of header files, dynamically linkable libraries, Java class files, and an extensible runtime environment. This distribution will be (not yet implemented) available on the WWW in platform-specific installation formats (e.g. SGI's tardist image and Window's installation wizard). Also, a mailing list has been established for interested developers to freely exchange comments. There will be no licensing fee or shareware charge. Plans are being made to provide ongoing support and maintenance; developments will be announced on the mailing list, as they become known. Additional papers and information may be found at http://npsnet.nps.navy.mil/Bamboo.

10: Conclusions

Bamboo overcomes many common VE system architecture pitfalls by providing modular components and a dynamically extensible runtime executable. These two features enable a flexible framework on which a variety of applications may be written.

Although it is hoped to be a significant contribution to the VE community, only large-scale adoption will reveal how the system responds to widespread use. Furthermore, only through the widespread use of a common framework can the efforts of unrelated groups be integrated seamlessly, which is imperative if truly robust networked VEs are ever to be achieved.

Acknowledgements

Bamboo has evolved over time as the result of the efforts of the main author and colleagues Joel Brand and Andrzej Kapolka,. Furthermore, the patience of Dr. Mike Zyda and the NPSNET Research Group has been appreciated. Finally, this effort could not have been without the generous support of our sponsors: DARPA, ONR, and ANS.

References

[1] Adobe (1997). Photoshop Software Development Kit, ftp://ftp.adobe.com/pub/adobe/devrelations/sdk/photoshop.

[2] Ben-Natan, R. (1995). CORBA : A Guide to the Common Object Request Broker Architecture, McGraw Hill Text.

[3] Bricken, W. and G. Coco (1994). "The VEOS Project." Presence **3**(2): 111-129.

[4] Carey, R. and G. Bell (1997). The Annotated Vrml 2.0 Reference Manual, Addison-Wesley.

[5] Carlsson, C. and O. Hagsand (1993). "DIVE - A Platform For Multi-User Virtual Environments." Computer and Graphics **17**(6): 663-669.

[6] Coryphaeus Software (1997). EasyScene, http://www.coryphaeus.com/products_dir/es.html

[7] Deering, M. and H. Sowizal (1997). Java 3D API Specification, Addison-Wesley.

[8] Deline, R. (1993). Alice: A rapid prototyping system for three-dimensional interactive graphical environments. Computer Science Department. Charlottesville, University of Virginia.

[9] Division (1997). dVISE, http://www.division.com/5.tec/a_papers/uvp.htm.

[10] Green, M., C. Shaw, et al. (1993). Minimal Reality Toolkit. Department of Computer Science, University of Alberta.

[11] Greenhalgh, C. and S. Benford (1995). MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading. Distributed Computing Systems (DCS'95), Vancouver, Canada, IEEE Computer Society.

[12] Horstmann, C. S. and G. Cornell (1997). Core Java 1.1 : Fundamentals, Prentice Hall Computer Books.

[13] Macedonia, M. R., M. J. Zyda, et al. (1994). "NPSNET: A Network Software Architecture for Large Scale Virual Enivornments." Presence **3**(4): 265-287.

[14] Microsoft (1997). Direct 3D Immediate Mode.

[15] Microsoft (1997). Direct3D Retained Mode.

[16] Musser, D. R. and A. Saini (1996). STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library. Reading, Massachusetts, Addison-Wesley Publishing Company.

[17] Neider, J., T. Davis, et al. (1993). OpenGL Programming Guide, Addison-Wesley.

[18] Netscape (1997). Navigator 4.0 Plug-in Guide, http://developer.netscape.com/library/documentation/communicator/plugin/contents.htm

[19] ObjectSpace (1997). Systems<Toolkit>, http://www.objectspace.com/toolkits.

[20] Paradigm (1997). Vega, http://www.paradigmsim.com/vega.html.

[21] Rohlf, J. and J. Helman (1994). IRIS Performer: A Hign-Performance Multiprocessing Toolkit for Real-Time 3D Graphics. SIGGRAPH'94.

[22] RogueWave (1997). The Total Solution Suite, http://www.roguewave.com/products/products.html.

[23] Schmidt, D. (1993). The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications. 11th and 12th Sun Users Group. http://www.cs.wustl.edu/~schmidt/SUG-94.ps.gz

[24] Sense8 (1997). WorldToolkit, http://www.sense8.com/products/worldtoolkit.html.

[25] Silicon Graphics (1997). OpenGL Scene Graph / OpenGL++, http://www.sgi.com/cosmo/cosmo3d

[26] Singh, G., L. Serra, et al. (1994). "BrickNet: A Software Toolkit for Network-Based Virtual Worlds." Presence **3**(1): 19-34.

[27] Snowdon, D. N. (1994). "AVIARY: Design Issues for Future Large-Scale Virtual Environments." PRESENCE **3**(4): 288-308.

[28] Sridhar, M. A. (1995). Building Portable C++ Applications with YACL, Addison-Wesley.