Behavior-Friendly Graphics

Kenneth B. Russell and Bruce M. Blumberg
The Media Lab, MIT, 20 Ames Street, Cambridge, MA 02139
{*kbrussel, bruce*}*@media.mit.edu*

Abstract

Interactive autonomous and directable characters require certain architectural support such as a motor system which moves the character based on decisions made by the behavior system. While previous work has focused on these higher-level systems, we focus on the infrastructure provided by the graphics system. We describe how a graphics system designed with synthetic characters in mind can be behavior-friendly, *simplifying motor and behavior system construction and making new characters easier to create. The system described in this paper was the underlying support for "Swamped!", an interactive 3-D cartoon demonstrated in the Enhanced Realities section of SIGGRAPH 98 [6, 8]. We describe how our design and implementation allowed us to achieve real-time performance in this Java application.*

Figure 1. A scene from Swamped!, in which the chicken has just lured the raccoon onto the bullseye with a fake egg obtained from the Acme truck. See also the color plates.

1 Introduction and Motivation

Most real-time interactive animated character systems [4, 10] sport a three-tier architecture: the *behavior system*, responsible for action selection; the *motor system*, responsible for turning high-level commands from the behavior system into motion of the character; and the *graphics system*, responsible for rendering and managing the character's geometry.

Previous work in the synthetic character domain has focused largely on the behavior and motor systems. Our group recently moved to Java from a C++-based environment. Because of a lack of portable, high performance 3-D graphics libraries for Java at the time, we needed to develop a graphics layer that would shield the motor and behavior systems from the specifics of the underlying graphics system (in our case Cosmo3D and Performer) and deliver the highest performance possible, since our experience with Java suggested that a naive implementation would not take full advantage of the performance of the underlying graphics system. We did not use Java3D at the time because it was still under development and not available on the SGI platform; however, the lessons are equally applicable to an implementation based on Java3D, or indeed any graphics library. As we revisited how motor and behavior systems typically use a graphics system, we made several useful discoveries about what functionality is necessary, how the graphics system can simplify behavior system construction, and how to achieve real-time 3-D graphics performance within Java.

A synthetic character system has certain domain-specific characteristics which suggest design decisions for the graphics system:

- **Abstraction of resources.** As demonstrated by Blumberg and others [4, 11], motor systems typically use abstractions such as articulated variables (avars) or degrees of freedom (DOFs) to modify underlying graphics resources such as transforms. More generally, DOFs may need to handle joints, inverse kinematics, and meshes. The desire to make very different types of graphics resources appear the same, and to handle multiple implementations of the motor system with minimal reimplementation, suggest the abstraction level should be moved into the graphics system rather than remaining in the motor system.

- **Object-object interaction.** The system must support inter-character interaction, primarily collision detection and response and grabbing of one character by another. Typically, such interaction requires computing local-to-world transforms for kinematic chains. As a result, an efficient joint-link model is needed.

- **Limited scene graph access.** As a result of using abstractions such as avars or DOFs, only a few hooks are needed into the underlying scene graph. More generally, the scene graph is only accessed in certain well-defined and limited ways once it is constructed.

- **Unidirectional data flow.** A character's motion comes either from the character itself (using either procedural or artist-animated approaches) or from the world (the application-side object which manages inter-character interaction such as collision detection). Even in the case of collisions, characters often respond in character-specific ways. Thus, decisions of how a character moves are almost always made above the graphics layer. This implies that data flows unidirectionally from the application down to the graphics library at run time.

- **Cost of communication.** Touching the scene graph is potentially expensive for many reasons: the scene graph may perform internal updates when writes occur, the native code interface may be slow, and the scene graph may be running in another process or on another machine. The fundamental assumption is that the underlying scene graph API is native (written in C++) and/or running in a separate process; either way there is a cost to crossing that boundary.

- **Portability and configurability.** The graphics system must target multiple platforms and scene graph implementations. It should take advantage of multiple processors or computers if available, and work equally well if running as a library or in a separate process. The distinction should be hidden from the motor and behavior systems.

- **Speed.** The graphics system must be fast. A character must not be constrained to six frames per second (FPS) performance if it requires thirty to express its personality.

These observations suggest the use of a *behavior-friendly graphics layer* that sits between the motor system and the native graphics system to implement much of this functionality and to take advantage of characteristics such as unidirectional data flow to increase performance.

The structure of the rest of the paper is as follows. Section 2 discusses related work and the specific contributions of this paper. Section 3 defines our terminology and describes avars, DOFs, and the joint-link model. Section 4 describes the fundamental functionality the graphics system must provide and how our system implements it. Section 5 discusses the issue of state management, which is crucial to high performance. Section 6 describes our application and its performance.

As a brief introduction to the discussion below, a character's body in our system is ultimately represented in the underlying graphics library as a scene graph composed of implementation-specific nodes such as transforms, shapes, and materials. We use standard modeling packages such as 3D Studio Max 2 to model our characters and export the geometry as VRML 2 files. At load time these files are read and the underlying scene graph is constructed. In addition, the graphics layer instantiates the objects that the higher layers of the system (e.g., the motor and behavior systems) will need in order to modify the underlying geometry at run time. The most important of these objects are described below.

2 Related Work

Architectures of virtual reality systems are most closely related to this work. The early work of Zeltzer et al. [21] describes the integration into an interactive framework, and application of, modules controlling, for example, user input, inverse kinematics, and dynamics. Appino et al. [1] run such modules on independent computers and use asynchronous communication to avoid round-trip network delays. Shaw et al. [15] decouple the application similarly, and in addition allow rendering of the scene independently of the computation of the underlying simulation. Later work addresses the issue of reducing lag in such multiprocessor situations. Wloka [20] describes a model for estimating lag and a “just-in-time” synchronization approach to minimize it by scheduling processes at the proper times. Jacobs et al. [7] perform just-in-time data acquisition at the appropriate moment in the computation loop to minimize end-to-end lag, and also extrapolate tracking information from their input devices.

The Menv system described by Reeves et al. [11] breaks up a modeling and animation system into a set of tools which run in separate processes and communicate via shared memory. While interactivity is important in this application, this system focuses on enabling the artist to create new animations for characters rather than on generating real-time output.

In the domain of interactive animated characters, Perlin et al. [10] consider the distribution of characters across both local- and wide-area networks. They run characters' behavior engines on separate computers, rendering from a dedicated machine. Russell et al. [12] applied a similar distributed computation scheme to Blumberg's behavior system architecture.

The focus of this work differs from the above in several ways. Much of the virtual reality work focuses on the necessity of increasing throughput and decreasing lag for the purpose of making a head- or hand-tracking system more responsive. Our application is much more forgiving of lag, as the user interacts through high-level gestures [8], and we use a projection display for output. Many of the VR applications described in the above papers do not deal with hierarchically structured objects such as characters with limbs. However, the paradigm we describe of separating the application (such as physical simulation) from the graphics system is present in most of these systems, usually out of necessity; most of these VR systems are implemented in heterogeneous environments of networked single-CPU computers. We go one step further and specifically describe how unidirectional data flow may be attained between our application and graphics subsystem and why this is important for applying parallelism and increasing performance.

Menv addresses many of the same issues as this work (grabbing, for example) and also describes an architecture for implementing an authoring tool for computer animation. This work focuses instead on the graphics support required to implement animated characters with behavior; a tool like Menv might be used to author individual animation segments for characters in a system such as ours.

While earlier work in the interactive character domain has considered the problem of distributing computation and increasing throughput, this work has three specific goals not addressed by these earlier works: to describe the graphics-level support needed by a character with behavior, to show how implementing this functionality in the graphics layer rather than the motor or behavior systems simplifies behavior system construction, and to show how this functionality can be implemented for good performance on both single- and multi-processor systems, providing real-time (30 FPS) performance in a Java application performing 3D graphics in a multi-processor configuration.

3 Avars, DOFs and the Joint-Link Model

We borrow from Reeves et al. [11] the term *articulated variable*, or *avar*, to indicate the abstraction of a graphics resource. Avars can map directly to entities in the scene graph: for example, a `Transform` node could have rotation, translation, and scale avars created for it, depending on which fields of the transform the character's animations are actually modifying. Avars can also map to more abstract entities like poses of a deforming mesh. Each avar caches the value of its underlying resource to avoid performing a scene graph call when its value is queried. For this reason the uniqueness of avars is critical, and each node in the scene graph is responsible for creating avars for its fields. A `set` call on an avar sets only the cache, allowing the motor system to perform multiple `sets` during its update without sending redundant information to the graphics system. `Get` calls return the cached value without accessing the scene graph. A `sync` method initializes the avar's value from the scene graph state, and because all motion is generated by the character or the world, needs to be called only at the beginning of time. Changes are sent down via a `commit` method, which is potentially time consuming and is therefore done only once per frame, at the end of the character's behavior update. As all run time interaction with the scene graph is mediated through avars, it is crucial for performance reasons that the application not sidestep the avar mechanism (Section 5).

A *degree of freedom*, or *DOF*, is the motor system's interface to an avar, and adds a locking mechanism necessary for the motor system to arbitrate control of joints among conflicting motor skills. Our motor system is patterned after that described by Blumberg [4]. Since different implementations of the motor system might have different mechanisms for dealing with conflicting motor skills, the locking mechanism was left in the motor system, but the basic abstraction of the graphics resources was moved into the graphics system's avars.

The *joint-link model* organizes all of the avars for a character's geometric model into a hierarchy, to provide information about its current configuration. The joint-link model can be queried to obtain a world-to-local or local-to-world transform for any joint in the character. While this information is available in the scene graph's structure, using the avars' caches avoids scene graph calls when the joint-link model is queried. Using the joint-link model, grabbing functionality can be implemented entirely in the application, rather than requiring modifications of the scene graph's structure (Section 4.3).

4 Core Functionality

4.1 Kinematic Animation

A graphics system for synthetic characters must first and foremost support forward kinematic animation. Since most scene graph APIs support hierarchical transforms, the implementation of kinematic animation in the graphics system requires only the creation of avars for joints which were animated by the artist beforehand. We deduce which joints are in use by examining the content of VRML 2 files which specify the character's motion, allowing us to export animations from off-the-shelf packages directly into our system.

An artist-generated animation comes into our system as a series of keyframes specifying the orientation of each joint. As the animation plays, the two keyframes closest to the current time are determined, and an `Interpolator` object computes the joint's state, typically using spherical linear interpolation [18].

4.2 Mesh Animation

A *mesh animation* is a set of poses. Each specifies the vertex positions for a piece of geometry and has an associated alpha value, typically between 0.0 and 1.0. This mechanism can be used to perform facial animation as described by Perlin [10]: for example, "0.7 happy, 0.3 sad". It can also be used to animate deforming meshes through longer sequences such as a walk cycle; in this case each alpha value corresponds to a keyframe in the animation.

Rather than expose the vertices of the geometrical object to Java, we created a `MeshAnim` class which allows the Java application to set per-keyframe alpha values. Vertex positions are interpolated in C++ code, drastically reducing the amount of data sent from the application to the native graphics library. A `MeshAnim` might contain keyframes for several underlying pieces of deforming geometry (for example, the body and feet of a chicken; see Section 6), and further, contain several animations for all of these geometries (for example, walk, hop, and fly.) Each alpha value corresponds to a pose of all the underlying deforming geometries; note that we require that all animations loaded into a `MeshAnim` animate the same set of geometries. The alpha values for all of the animations are concatenated linearly, so the `MeshAnim`'s value is an array of floating point numbers. A `MeshAnimAvar` provides a cache for this array.

In order to make mesh animations appear the same to the motor system as kinematic animations, we created a `MeshAnimInterpolator` which keeps track of the range of alpha values in the underlying `MeshAnim` corresponding to a particular animation. When the interpolator's value is set to a value between 0.0 and 1.0, it determines the two adjacent keyframes to the current position in the animation and sets their alpha values to blend between the two. We currently interpolate linearly between adjacent keyframes for mesh animations, although more sophisticated interpolation schemes could be used to provide better results.

4.3 Grabbing

Grabbing conceptually enforces a constraint: one character grabs another, attaching the grabbed character to the grabber's end effector. In our system, grabbing is implemented using the joint-link model (Section 3), which provides the local-to-world transform for the end effector. This is then composed with the grabbed character's current desired orientation and an optional offset transform to determine its new position each frame.

A similar result could be obtained by reparenting the geometry of the grabbed character in the scene graph, which would require less per-frame computation. This approach would, however, expose the scene graph structure to the Java application, reducing portability. In addition, it would require that the geometric structure of the character correspond directly to the hierarchy in the scene graph. This correspondence does not hold for characters animated entirely with `MeshAnim`s, because one piece of geometry (the "skin") corresponds to several pieces of rigid geometry connected by transforms in a kinematically animated character. We allow `MeshAnim`-animated characters to grab others by animating a set of transforms corresponding to the character's hierarchy simultaneously with the deforming mesh. These transforms contain no geometry, however.

ALIVE [3] implemented grabbing as a motor skill. Pushing this functionality down into the graphics system makes creation of new characters simpler because the default grabbing behavior "does the right thing" with no additional implementation work. Both grabber and grabbee are notified that a grab has begun or ended and can make behavior-level decisions on this information, so customization, such as playing a squirm animation upon being grabbed, is also possible.

4.4 Collision Detection and Response

Rather than perform full polygon-polygon collision detection, we chose to use simplified bounding volumes. Our system currently supports spheres and oriented bounding boxes that can either be specified by the artist or computed automatically by the graphics system. The bounding volumes are updated by the application rather than the native graphics layer to maintain unidirectional data flow. Collision response can be turned on and off on a per-character basis, and we support VRML-style "proximity sensors" which are collidable objects that do not induce a collision response in the character.

When a collision is detected between two characters, the behavior system of each collided character is notified. Characters can thereby react to collisions in more sophisticated ways than following the simple "no interpenetration" rule enforced by the graphics system. For example, the houses in our system wiggle when they are collided against to indicate that they are live characters with which one can interact.

Support for collision detection in the graphics system, combined with a link to the behavior system, vastly simplifies behavior system construction. ALIVE, for example, required behaviors to be added to each character for proximity detection and collision response.

5 Achieving High Performance

5.1 Smart State Management

As mentioned in Section 1, we make the fundamental assumption that accessing the scene graph is an expensive operation, for writes as well as for reads. The application therefore needs to be clever about where it stores its state to minimize calls down into the native graphics layer.

The avar mechanism provides caches for all values being read or written in the scene graph below, with the following two results: first, redundant writes to a particular avar do not cause redundant communication with the native graphics layer, since avars are written once per frame when their `commit` method is called. Second, and more importantly, *all* "get" calls to the scene graph are avoided at run time. Gets are especially expensive because they require the application and native graphics layer to synchronize. When the native graphics layer is embedded in the same process as the application, as when it is used as a library, this synchronization is

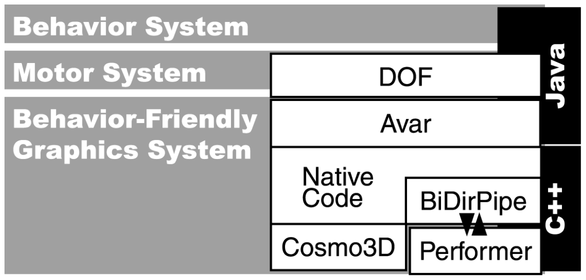

Figure 2. System architecture, indicating links across which parallel processing can occur.

irrelevant. However, when the native graphics layer is run in parallel, as on a multiprocessor or multi-computer configuration, synchronization is disastrous to performance. To make the caches work properly, it is crucial that the application not sidestep the avar mechanism and deal with the scene graph directly.

5.2 Parallelism Support

On a uniprocessor machine a serial implementation of the native graphics layer is most efficient. However, running the native graphics layer in parallel to the application can provide a significant speed improvement when another processor or computer is available. Unidirectional data flow at run time is the primary property of the system which makes it amenable to parallelization. This property came about because of the nature of the graphics system's design; for example, complex scene graph reorganization and geometry creation at run time were prohibited. While it may seem that the constraints imposed by this design were too restrictive, in practice we found that the system handled all of the operations we needed to build our characters, and the speed improvements afforded by a parallel implementation more than made up for the initial adjustment to our interface design from that of a more general scene graph API.

Our parallelization approach partitions the program into two distinct pieces: the application (written in Java), and the graphics (written in C++). The strong distinction between the two is indicated in Figure 2. The application and graphics

```
class BiDirPipe {
public:
    virtual bool read(void *dest, int size) = 0;
    virtual bool write(void *dest, int size) = 0;
    virtual bool poll() = 0;
    virtual void flush() = 0;
    virtual void lock() = 0;
    virtual void unlock() = 0;
};
```

Figure 3. BiDirPipe base class.

programs run either in separate processes on the same computer or on two different computers connected by a network. In both configurations all of the communication between Java and the graphics program occurs via a small native class called a `BiDirPipe` (Figure 3), which is implemented either with shared memory [17] or a socket [16].

The simple protocol used to communicate between the Java application and the graphics process is similar to that in a Remote Procedure Call [16] mechanism; see Appendix A for details. All graphics API methods returning `void`, most significantly rendering, are done in parallel to the main application. The structure is essentially a software pipeline for the graphics system on top of that already implemented by the scene graph API.

An early comparison between serial and parallel bindings of Cosmo3D on a multiprocessor SGI Onyx2 showed an increase from 19.2 FPS for the serial implementation to 31.25 FPS for the parallel, a 50% speed improvement. Note that this did not require any changes to the underlying graphics library, but merely changed the interface to it.

6 Results

We developed an interactive 3D cartoon, "Swamped!" [6, 8], in which the protagonist, K.F. Chicken, thwarts the raccoon's attempts to steal his eggs; see Figure 1 and the color plates. Swamped! was demonstrated as part of SIGGRAPH 98's Enhanced Realities exhibition. The chicken is a directable character, controlled by the user via a plush toy instrumented with sensors, while the raccoon is fully autonomous. The raccoon is a standard hierarchical model with 43 rotational joints and a face morphed among six primary facial expressions using mesh animation; the model contains 8000 triangles. The chicken is a combined hierarchical model and deformable mesh; its body and feet are animated solely with mesh animation, while its head and wings are rigid pieces of geometry which have squash and stretch added with non-uniform scales. The chicken has over 600 keyframes for all of its body animations (for example, walk, jump, and run), and contains 3500 triangles. Three houses animated with mesh animation total roughly 6000 triangles.

Swamped! has been run on a two-processor 400 MHz Pentium II PC using Cosmo3D, on an 8-processor Silicon Graphics Onyx2 using Performer, and in a hybrid mode in which the Java application runs on a PC, communicating over 100Mbps Ethernet to the Performer-based graphics system running on the Onyx2. Because the Java just-in-time compiler is highly optimized on the PC, the latter is the fastest configuration: the per-frame behavior and motor system updates take roughly 20 ms, while the writing of the current frame's graphics commands to a socket takes 10 ms. We achieve sustained 30 FPS performance because all rendering is done in parallel, no round-trip queries are made to the graphics system by the Java application at run time, and the underlying graphics library is fast enough to keep up with the Java application. The limiting factor is the communication overhead.

7 Conclusion

We have presented a graphics system which is "behavior-friendly", designed with synthetic characters in mind. We have described the criteria which the graphics system must meet, the structure which the synthetic character domain suggests, and the advantages, such as unidirectional data flow, that structure provides. We have discussed kinematic and mesh animation, grabbing, and collision detection and response in the context of our graphics system, and have shown how moving higher-level concepts into the graphics layer simplifies behavior system construction, making new characters easier to design. Finally, we have shown that using a parallel approach it is indeed possible to achieve real-time 3-D graphics performance in a non-trivial Java application.

8 Acknowledgments

We thank the rest of the Swamped! team: Michal Hlavac (who also drew Figure 2), Christopher Kline, Michael P. Johnson, Teresa Marrin, Bill Tomlinson, Andrew Wilson, Song-Yee Yoon, and Marc Downie; our UROPs who helped on this project as well as Swamped!: Jeremy Lueck, Dan Stiehl, Zoe Teegarden, and Jed Wahl; and Dr. Joseph Paradiso for the chicken's sensor systems.

References

[1] P. A. Appino, J. B. Lewis, L. Koved, D. T. Ling, D. A. Rabenhorst, and C. F. Codella. An architecture for virtual worlds. *Presence*, 1(1):1–17, 1992.

[2] David Beazley. Simplified wrapper and interface generator. 1995. `http://www.swig.org/`.

[3] Bruce Blumberg. *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, Massachsetts Institute of Technology, 1996.

[4] Bruce Blumberg and Tinsley Galeyan. Multi-level direction of autonomous characters for real-time virtual environments. In *Computer Graphics Proceedings, SIGGRAPH 95*, pages 47–54. ACM, 1995.

[5] Jürgen Döllner and Klaus Hinrichs et al. Modeling and animation machine/virtual rendering system. 1998. `http://wwwmath.uni-muenster.de/˜mam/`.

[6] Bruce M. Blumberg et al. Swamped!: Using plush toys to direct autonomous animated characters. In *SIGGRAPH 98 Conference Abstracts and Applications*. ACM, 1998.

[7] M. Jacobs, M. Livingston, and A. State. Managing latency in complex augmented reality systems. In *Proceedings, Symposium on Interactive 3D Graphics*, 1997.

[8] M. P. Johnson, A. Wilson, B. Blumberg, C. Kline, and A. Bobick. Sympathetic interfaces: Using a plush toy to direct synthetic characters. In *CHI 99*, 1999. To appear.

[9] Sheng Liang. *Java Native Interface: Programming Guide and Reference*. Addison-Wesley, 1998. `http://java.sun.com/products/jdk/1.1/docs/guide/jni/`.

[10] Ken Perlin and Athomas Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *Computer Graphics Proceedings, SIGGRAPH 96*, pages 205–16. ACM, 1996.

[11] William T. Reeves, Eben F. Ostby, and Samuel J. Leffler. The Menv modelling and animation environment. *Journal of Visualization and Computer Animation*, 1(1):33–40, August 1990.

[12] K. Russell, B. Blumberg, A. Pentland, and P. Maes. Distributed alive. In *SIGGRAPH 96 Technical Sketches*. ACM, 1996.

[13] Kenneth B. Russell. An automatic C++ to Scheme interface generator. 1995. `http://www.media.mit.edu/˜kbrussel/Header2Scheme/`.

[14] Kenneth B. Russell. A Scheme binding for Open Inventor. 1995. `http://www.media.mit.edu/˜kbrussel/Ivy/`.

[15] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the mr toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, July 1993.

[16] W. Richard Stevens. *UNIX Network Programming*, chapter 4. Prentice Hall, 1990.

[17] W. Richard Stevens. *Advanced Programming in the UNIX Environment*, chapter 14. Addison-Wesley, 1992.

[18] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques*, chapter 15. Addison-Wesley, 1992.

[19] Jeff White. A Java binding for Open Inventor. 1996. `http://www.igd.fhg.de/CP/kahlua/`.

[20] Matthias Wloka. Lag in multiprocessor virtual reality. *Presence*, 4(1):50–63, 1995.

```
void Java_graphics_Transform_setTranslation(
  jobject javaTransform,
  jfloat x, jfloat y, jfloat z) {
   Transform *xform =
    ExtractCPlusPlusTransformFromJavaTransform(
     javaTransform);
   xform->setTranslation(x, y, z);
}
```

Figure 4. Serial mechanism for glue code, illustrating the mapping of the `setTranslation` method of a `Transform` class from Java to C++.

[21] D. Zeltzer, S. Pieper, and D. Sturman. An integrated graphical simulation platform. In *Graphics Interface '89*, pages 266–74, Toronto, Canada, 1989. Canadian Inf. Process Soc.

A Interface Specifics

There are two ways to bind a native C++ library into an interpreted language: a standard, serial method, and our new, parallel method. This section compares the two and describes the specifics of our parallel interface. We restrict our example to Java and the Java Native Interface [9] for calling C++ code from Java, although our method could be applied to any language with a foreign function interface.

Methods in a Java class can be specified as "native", meaning the interpreter will call a C function rather than a Java method. A C++ library can be bound into Java by creating a Java-side wrapper class for each C++ class, which contains native methods analogous to those in the C++ class. Each of these native methods turns around and calls the method on the underlying C++ object. Figure 4 illustrates the standard mechanism for this Java to C++ mapping, which we use for our uniprocessor graphics library binding. There are tools available to generate such *glue code* automatically [13, 2], and all currently available 3D API bindings for interpreted languages (for example, Ivy [14], Kahlua [19], and MAM/VRS [5]) follow this pattern.

To allow parallel processing of graphics calls, we explicitly split the C++ graphics layer into its own process, and manage communication with the Java process with a `BiDirPipe` (Figure 3). All Java calls to `set` methods on graphics objects correspond to a write on this pipe. All calls to `get` methods must write the request to the pipe, flush it, and wait for the graphics process to respond. Asynchronous writes from the graphics process back to Java are not allowed in our protocol. Figure 5 illustrates the Java side of this communications mechanism. In the graphics process, a dispatcher reads the message ID and calls a function which reads the rest of the message and calls the appropriate method (`setTranslation`) on the C++ object. For clarity, byte swapping macros are not included in this example, but are required when passing elementary types such as ints and floats between little-endian and big-endian architectures; see Stevens [16] for details. Our convention, as in many RPC implementations, is to send only big-endian data over the pipe.

```
typedef enum {
  TRANSFORM_SET_TRANSLATION,
  ...
} GraphicsMessageId;

typedef struct {
  // For example, ''TRANSFORM_SET_TRANSLATION''
  int messageId;
  // For the C++ object
  void *nativePtr;
  // New values for
  // the transform's translation
  float x, y, z;
} MessageV3f;

void Java_graphics_Transform_setTranslation(
  jobject javaTransform,
  jfloat x, jfloat y, jfloat z) {
   BiDirPipe *pipe =
    ExtractBiDirPipeFromJavaGraphicsObject(
     javaTransform);
   MessageV3f message;
   message.messageId =
    (int) TRANSFORM_SET_TRANSLATION;
   message.nativePtr =
    ExtractCPlusPlusTransformFromJavaTransform(
     javaTransform);
   message.x = x;
   message.y = x;
   message.z = x;
   pipe->lock();
   pipe->write(&message, sizeof(message));
   pipe->unlock();
}
```

Figure 5. A complete example of the Java side of our parallel processing glue code mechanism. All of the C++ code in the `setTranslation` method is executed in another process, in parallel to the Java code which called this native method.

This parallel glue code structure requires a significant amount of additional mechanism over the standard serial structure. In addition, it is best suited for applications which are cleanly divisible into modules which have unidirectional data flow, and must be accompanied by Java-side caches to avoid round trip calls to the remote process. Despite the extra effort required to implement it, this new glue code structure provides significant advantages over the serial version. The ability to distribute the computation and the associated performance increase is primary among these. Debugging of the graphics process is made easier, since it is not embedded in the Java process. In addition, graphics calls may be made in multiple Java threads regardless of whether the underlying C++ graphics library is thread-safe, since communication is mediated and serialized by the `BiDirPipe`.