by
Cary Sandvig (Cary.Sandvig@disney.com)
and
Jesse Schell (Jesse.Schell@disney.com)

Introduction

Panda3D (Platform Agnostic Networked Display Architecture) is an open source software architecture for rapid development of networked real-time 3D applications. Its key features are:

- Powerful scenegraph semantics
- Abstraction layers for portability
- Rapid prototyping enabled through late-binding scripting languages
- Easy-to-use network architecture for rapid creation of shared worlds

Panda3D is available for download from www.panda3D.org.

Scenegraph Design

The design of Panda3D was driven by desires for both the ability to capture the semantics of a model cleanly, and to have a thin abstraction for the actual hardware. Many of these decisions were realized in what we call “noun-verb separations,” an attempt to avoid mixing actions or behaviors with static data objects.

This led to a departure from standard practice in the scene-graph design. Commonly, information such as local transformation, color, and texture is stored directly on the nodes of the graph. This typically leads to either very large nodes (to accommodate all possible state changes that may be desired), or a very large number of node types (to provide all the combinations of state change). Instead, we view the arcs of the scene graph as first-class objects which represent some change in state. The nodes, then, represent being in a particular state. Our nodes become very light-weight, holding only actual geometry, or acting as a handle to deal with higher-level structure.

This approach not only simplifies our representation, but also facilitates scene graph inquiries that were difficult before. For instance, a standard request asks how two nodes in the scene graph are related to one another spatially. Now we can ask about such relationships over any state change, such as color, texture, or fogging. This structure has also given us, for example, an extremely flexible instancing mechanism that allows us to alter much more than just the transformation leading to each instance.

The Panda3D scene graph is a specialized instance of a general-purpose graph representation and handling mechanism. Often, many special purpose graphs will exist within a single application, and some nodes may participate in multiple graphs. For example, a sample application in the Panda3D distribution constructs a "data graph" which describes and controls how input from the user is transformed into camera movements and actions on the scene graph. Alternatively, a culling pass can be made on the scene graph, building a new graph on those nodes that represents only what is visible, thus reducing what the drawing pass must traverse.

Other Abstraction Layers

Another part of the design that was guided by the "noun-verb separation" principle is in the Graphics State Guardian (GSG). The GSG plays multiple roles in the Panda3D system: abstraction of the rendering hardware, filter for unnecessary or wasteful state change, and "verb" for drawing the "noun" of the scene graph. This factoring enables the use of many different back-end rendering libraries, even in the same application. New implementations of the GSG can be added without ever having to touch scene graph code. Furthermore, the GSG specifies exactly what functionality needs to be present (or emulated) on a given platform.

Similar thin abstraction layers exist or are in development for other functions, such as: sound, networking, process/threading, windowing, device I/O, etc. Since Panda3D only calls these functions via abstraction layers, third-party libraries (VRPN or NSPR for example) can be used by Panda3D. These libraries are wrapped with an adapting interface. In this way we can leverage the work done in these areas without having changes in these packages affect the Panda3D code that uses it.

Interactive Scripting

Efficiency and rapid prototyping are two keys to creating high quality virtual worlds. Unfortunately, no single computer language adequately provides both. C++ is the coin of the realm for efficiency, and for access to third-party API's. However, it is an early-binding compiled language, which prevents programmers from making changes to their software while a simulation is running.

Late binding “scripting” languages, such as Scheme, Python and Smalltalk are excellent for rapid prototyping, but lack the necessary efficiency and API access necessary to create detailed virtual worlds.

Interrogate

Panda3D gives the programmer the best of both worlds by providing a way to use both C++ and an interactive scripting language simultaneously. Any language that has a foreign function interface allowing calls to C libraries can make use of Panda3D’s Interrogate system.

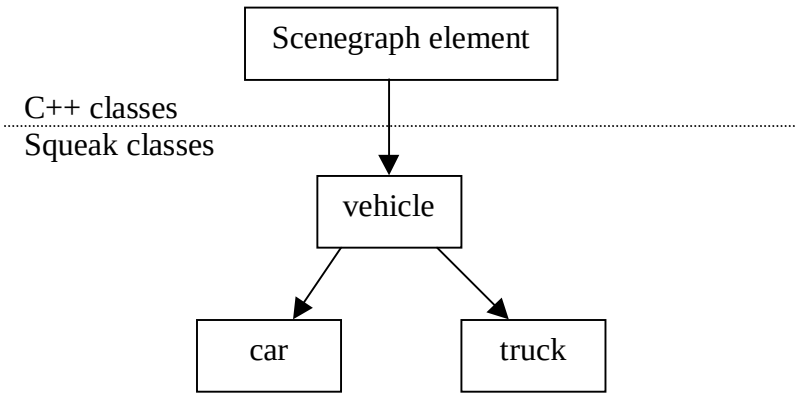
Interrogate works like a compiler, scanning and parsing C++ code. Instead of creating object libraries, it creates an “interrogate database” of objects, methods, global variables, etc. that are contained in the corresponding C++ library. This database may be queried to discover these functional elements and their interfaces.

To make use of this database, one creates an automatic code generator in the scripting language that scans the interrogate database and generates scripting language wrapper calls that execute the C library calls, via the scripting language’s foreign function interface. A benefit of this system is that interrogate imposes no restrictions on exactly how a scripting language interfaces with the C libraries, allowing them to interface in whatever way best “fits in” with the natural environment of a particular language.

Squeak

Panda3D includes an automatic code generator for Squeak (an open source Smalltalk implementation: www.squeak.org) because it is our favored scripting language. Squeak is an entirely object-oriented language, although its syntax, rules, and implementation are vastly different from C++. Nonetheless, we have been able to create a Squeak interface to C++ that allows object-orientation to cross the boundary between the languages. For each C++ class, a “proxy class” is automatically generated in the Squeak code, which exposes all of the public functions of the C++ class.

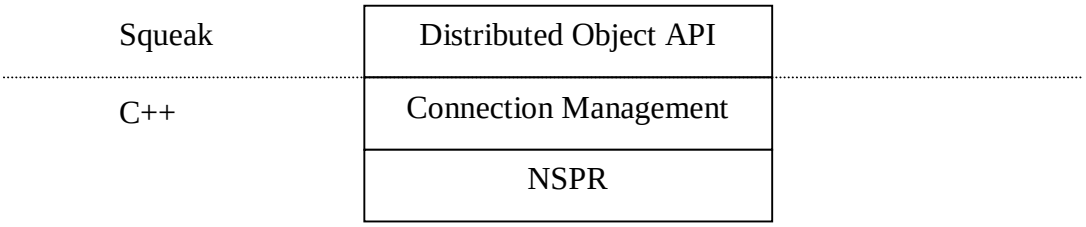
This not only allows for easy access to (and debugging of) C++ classes, but also allows for Squeak methods to be added to C++ classes, and for Squeak classes to appear to inherit from C++ classes. For example, a 3D driving simulation
might include vehicle classes which inherited from the “scenegraph element” class. The cross-language inheritance hierarchy might look like the following:



This is particularly useful, since each car or truck has all the interfaces that the efficient C++ scene graph provides, seamlessly integrated with the flexible, reprogrammable-on-the fly interfaces that Squeak classes provide.

Rapid Prototyping of Shared Worlds

The Panda3D support for rapid prototyping of shared worlds is implemented through three layers:



From bottom to top, these are:

NSPR: The Netscape Portable Runtime layer gives portable, robust, open source interface to Posix threads and TCP/UDP sockets. See http://www.mozilla.org.

Connection Management layer: A robust set of C++ classes forming an API that hides most of the details of sockets, byte streams, and threading, while giving the programmer the flexibility to manage arbitrary sets of TCP/UDP connections, and handle them in either single or multi-threaded fashion. Efficient clients, servers, and peer-to-peer solutions can easily be crafted on top of the Connection Management layer, either in C++, or in a scripting language via interrogate.

Distributed Object Layer: A simple system implemented in Squeak for rapid prototyping of shared virtual worlds. At the time of this writing (March 2000), the Distributed Object system is client/server based and is used as follows:

1. **Setup**
    a. *Start a server*
    The server can run on any machine (including a client machine). Panda3D includes a server implemented in Squeak and C++ that is useful for rapid prototyping because it makes it easy to debug problems between clients.
    b. *Create Distributed Object descendant classes*
    Any Squeak class which you would like to exhibit shared behavior needs to inherit from the DistributedObject class. It also needs to specify exactly which methods on it need to be distributed, what types they take, and in what manner they are to be distributed. This is a simple matter of configuring a table of data in one of the methods inherited from DistributedObject.

2. **Startup**
    a. *Create a ClientRepository*
    The ClientRepository is the class that interfaces to the Connection Management layer, and keeps track of all the Distributed Object instances currently in the system. In creating one, you specify all the DistributedObject descendant classes that you plan to use in the simulation, so that the ClientRepository can register them with the server.
    b. *Wait for Server Registration to complete*
    After the ClientRepository registers all the DistributedObject descendant classes with the server, the server then sends instructions to the ClientRepository about what instances of the DistributedObjects already exist in the world, and what their current state is. When this is complete, an event is thrown in the client that says that generation of new DistributedObject instances can now begin.

3. **Generate**
    When you generate an instance of a DistributedObject, the ClientRepositories on other clients are automatically notified, and automatically create duplicate instances of the DistributedObject.

4. **Update**
    When you update an instance of a DistributedObject, the ClientRepositories on other clients are also automatically notified, and update their local instance to reflect your changes. Updates can happen via TCP (guaranteed to arrive), or UDP (not guaranteed to arrive) as the programmer sees fit. The programmer may also separate the acts of the local update and distributed update to handle cases where, for example, it is desired that local updates happen very frequently, while the sending of remote updates happens only occasionally. Finally, the programmer may also specify whether a particular update is state-based (like a positional coordinate), and should be retained by the server, or event-based (like a chat message) and should not be retained by the server.

5. **Destroy**
    Destroying an instance of a DistributedObject causes the remote instances to be destroyed as well, and for the instance to be removed from the server. In addition,

if the server loses its connection with a particular client, any DistributedObjects that it has registered ownership for are automatically destroyed and removed from the database. To create an object that will not be destroyed in this way, a client must transfer ownership of it to the server.

6. **Locking**
   Every distributed object has a semaphore lock on it that controls who currently holds it. Locking is managed by the server, but not enforced. Enforcement is up to the clients. Any client may update any distributed object at any time, but to avoid conflicts in updating, a programmer may choose to use the locking mechanism so that clients can check whether it is acceptable for them to update a particular DistributedObject before doing so.

7. **Zones**
   A simple form of interest management works through numbered zones. When connecting, a client specifies what zone it is in, and the server only sends relevant generate and update commands. When a client changes zones, the server sends "disable" commands to that client for all DistributedObjects in the former zone, and "enable" commands (followed by necessary state updates) to that that client for all DistributedObjects in the new zone. In the current (March 2000) implementation, "disable" maps directly to "destroy", and "enable" maps directly to "generate". Future implementations are expected to include more complex caching schemes.

Conclusion and Future Work

This simple implementation of networking features has been very useful to us for rapidly prototyping shared virtual worlds. Other features currently in development include:

- A scalable server implementation with more complex interest management
- Peer-to-peer DistributedObject support
- Compressed and encrypted data streams
- Advanced streaming features to reduce bandwidth overhead

The features described here, as well as other features (such as soft- and hard-skinned character support, multi-pass rendering support, a real-time shader framework, etc), are the result of our work on previous systems used in the production of theme park and Location Based Entertainment (LBE) VR attractions. We have kept those ideas and tools that worked well, iterating on them to produce the current design. While there is certainly a cost associated with parts of the system, performance is very good and the increased flexibility afforded by the design offers an excellent trade-off.

For more (and more current) information about Panda3D, visit www.panda3d.org.