Looi™

a low-overhead object interpreter for MicroCosm™
by
Chip Morningstar

Lucasfilm Ltd. Games Division
November 10, 1985

INTRODUCTION

This document describes the design of **Looi™**, the **L**ow **O**verhead **O**bject **I**nterpreter, which forms the heart of the **MicroCosm™** host database system. The purpose of this document is to try to get it all straight before we get bogged down in implementation.

A note on notation: things in *italics* are new terms being introduced for the first time. The italics mean, ''Pay attention, this is a new term. Don't panic if you don't quite understand it. We are either just now explaining it or will explain it real soon.'' Things in**boldface** are proper names or symbolic identifiers for entities in the system.

OBJECTS

The host database is constructed out of *objects*. Objects are simply things with*pr operties*. Objects' behavior, use, and implementation are described below.

PROPERTIES

All objects have *properties*. An object's properties are simply a set of values that have meaning in the context of what sort of thing the object is supposed to be. These values may be integers, booleans, pointers to other objects, or more complicated structures, such as bitmaps or lists of things.

NOIDS

Each object is referenced by a *noid*, which stands for **N**umeric **O**bject **ID**entifier. Every object in existence has a unique noid. Noids are integers, but what precision integer remains undetermined. The constraint is that we may never be able to re-use a noid because we have no way to ever garbage collect them once they make their way into the outside world (well, perhaps we'll have a disintegrator object that causes other objects to cease existence, thus freeing their noids for re-use). It may be that the best way to do this is to simply use a big number (64 bits maybe), though that's hard on bandwidth. Perhaps variable-length humbers are the way to go. It doesn't really matter, as long as they are 1) compact in representation, and 2) can point at a lot of objects.

PROPERTY VECTORS

The database uses an object's noid to look up its *property vector*. This is an array of variable length (the length depends on what sort of object it is) that contains the information about the current state of the object's properties. Each element in this array is either the immediate value of a property (if the property is a simple integer or boolean value) or a pointer to a more complicated structure containing the relevant data in all its parts. What these elements are, what they mean, and how they are interpreted is all up to the software that actually uses the object.

ESSENTIAL PROPERTIES

The length of an object's property vector varies, depending upon the type of object it is. However, at a minimum all objects have the *essential properties* which are so called because they are essential to the

basic operation of the object system. The first few entries in the property vector are the essential properties. The property vector always contains at least these. There are currently four essential properties, though this may change as the design evolves. These four are*class*, *location*, *owner*, and *ima ge*.

CLASS

An object's *class* says what kind of object it is. The **class** property contains a pointer to a *class descriptor*, which is a data structure describing objects of a given type. In particular, this structure tells the number of properties that objects of the given type possess (i.e., how long the property vector is) and gives the object's *capabilities*. The**class** property is so important that we may deem it necessary to encode an object's class in the object's noid, and thereby avoid one level of pointer chasing in **Looi**'s inner loop.

LOCATION

An object's *location* says where in the world it is. The **location** property contains a set of coordinates in the **MicroCosm** coordinate system. See the document **MicroCosm Coordinate Systems and Topology**.

OWNER

An object's *owner* is the player entitled to manipulate it. The **owner** property contains a player account number.

IMAGE

An object's *image* is the way it appears when displayed graphically on a player's home computer. The **image** property contains a pointer to an *image descriptor*, which is a data structure describing an image, set of images, or animation sequence, in a reasonably display-device-independent fashion. See the document **MicroCosm Graphics**.

CAPABILITIES

An object may have *capabilities* which the player can exercise. Capabilities are things that the player can do with the object, such as move it, change its appearance, find out how much it weighs, etc. Capabilities are the means by which the home computers interrogate and manipulate the contents of the host database.

CAPABILITY PROCEDURES

The active part of a capability — the code which computes things and actually alters the contents of the database — is implemented by a *capability procedure*. This is a compiled procedure that runs on the host system. It takes a number of parameters which depend on the action that it is implementing. In any case it takes two parameters: the object whose capability is being exercised and the *capability number*. Any additional parameters are supplied by the entity requesting exercise of the capability.

CAPABILITY REFERENCE

A capability is referenced by specifying an object's noid together with a *capability number*. The noid is used to look up the object's property vector, from which is obtained the object's class descriptor. In the class descriptor is a pointer to a *capability vector*. The capability number is then used as an index into this vector to retrieve a pointer to the capability procedure. Depending on the architecture of the host processor, we may want to make the capability vector pointer the very first entry in the class descriptor, so that it will have a zero offset from the class descriptor address and thereby enable us to avoid adding the structure offset in the inner loop (in some processors we get this offset addition for free in the instruction operand fetch, so it won't matter, but in some it does not). The capability procedure is then invoked like any other procedure would be. Such a procedure lookup-and-invocation action is called a *capability reference*. The act of requesting a capability reference is called *exercising a capability*.

CAPABILITY PROCEDURE DEFINITION

Capability procedures are compiled in the native programming language of the host system. For purposes of discussion, I am assuming that this is **C**, but any other reasonably structured language (**Pascal**, **PL/1**, **Ada**, **Lisp**, etc.) will serve equally well. We will design a very simple specification language that we can implement (using **yacc** or a similar utility) as an equally simple **C** preprocessor that will let us lay out the object property vectors, class definitions and capability procedures in a tidy, compact form. Of course, we have come up with a terribly clever name for this language: **Noodl**, **N**egligable **O**verhead **O**bject **D**efinition **L**anguage. See the document also entitled**Noodl** for a complete description.

TRANSACTION STRUCTURE

The **MicroCosm** database access control structure is built around the concept of *transactions*. A transaction is an ''atomic'' exchange between **Looi** and the **Transaction Monitor** (see the **MicroCosm** system architecture block diagram). A transaction is basically a single capability reference. The **Transaction Monitor** is continually receiving requests from all the various remotes that are hooked into the host system at any given time. Its responsibility is to break these requests into individual transactions, serially feed these to **Looi**, and then route **Looi**'s responses back to the requesters. The mechanism by which is does this is beyond the scope of this document. See the companion document entitled **MicroCosm Host System Architecture** for details. The fact that transactions are fed to **Looi** serially is very important. The system never processes more than one operation on the database at a time. This is the means by which we avoid complicated locking, unlocking, and process synchronization procedures.

MAIN CONTROL LOOP

The main control loop of **Looi** is therefore simply a procedure which accepts a stream of incoming requests that are packaged as capability references. For each request, it looks up the appropriate capability procedure, calls it with the appropriate parameters, and returns whatever the capability procedure returned. It also must handle certain error checking (e.g., capability number out of bounds, etc.) and error response functions.

RIPPLE TRANSACTIONS

One of the things that a capability procedure can do is request further host system activity in the form of additional transactions with the database. Such transactions are called *ripple transactions* from the obvious image of transactions spreading out from one another like ripples in a pond. The most common use of ripple transactions is to cause players other than the one exercising a particular capability to be told about the result of the operation. This is done by initiating ''inquiry'' transactions on behalf of the players to be informed.

THE ASYNCHRONOUS NOTIFIER

Ripple transactions are fed to the **Transaction Monitor** by the **Asynchronous Notifier** (again, refer to the **MicroCosm** system architecture block diagram). It collects transactions being requested by capability procedures along with the identities of the players who should be notified of the results. In essence, it acts like another player talking to the **Transaction Monitor**, except that the responses go to a destination different from the source of the input. The detailed structure of the **Asynchronous Notifier** is described in the **MicroCosm Host System Architecture** document.