

Python for Massively Multiplayer Virtual Worlds

Jason Asbahr
Origin Systems, Inc.
jasbahr@origin.ea.com
jason@asbahr.com

Abstract

This paper presents a brief overview of the Python foundation of UO2*, the next-generation massively multiplayer virtual world from Origin Systems, Inc.

[* NOTE: development code name, final name to be announced]

1. Introduction

In the early 1990s, before the rest of the world heard about the Internet, the primary source of technical hype and excitement was “virtual reality”. Early virtual reality systems showed exciting promise, but most implementations were expensive, fragile, and immature [3]. In the mid-to-late 1990s, as the public began streaming onto the Internet, virtual reality was most often popularly associated with VRML, an attempt to merge the display technologies of virtual reality with the networking technologies of the Web [7]. Today, virtual reality lives on in the form of massively multiplayer virtual worlds, systems that allow hundreds of thousands of distributed users to participate in rich entertainment simulations. It is the interactive entertainment, or game, industry that acts as the primary driving force for innovation in this space.

UO2 is Origin System’s next-generation virtual world, an imaginary planet inhabited by all manner of fantastic, mythical, and technological creatures, places, and things. To enter this world, users run client software locally, connect to remote UO2 servers, and create one or more *player character avatars*. Player characters are highly customizable alternate identities who can walk around, talk, interact with objects, fight with other characters, learn new skills, and generally live life under the direction of the user. While characters start off life with basic skills, abilities, and equipment, they gradually become more powerful through exploration of the virtual world, battling monsters and opponents, and solving quests.

The new world of UO2 builds on the legacy of the swords and sorcery Ultima fiction originally created by Richard Garriott [1]. Like the original Ultima Online, UO2 brings thousands of players together in an artificial world. However, in addition to adding new chapters to the Ultima story, UO2 also expands on several technical fronts, enabling a system flexible enough for the rapid creation of many future virtual worlds. The system has an entirely new technological foundation of a robust server architecture and network protocol on the back end and an advanced real-time 3D graphics engine on the front end. Further, UO2 combines a highly data-driven design with simulation classes on both client and server, using the standard and well supported dynamic object oriented language, Python.

This paper will address three key ideas in the philosophy of the UO2 architecture. The first is the structure of the logical simulation domain; the second, the split between the maintenance of simulation reality and the presentation of that reality to users; and finally, the means by which the simulation utilizes a relational database for behavior definition and object state storage.

2. Simulation Domain

The Ultima universe is a reality similar to ours, approximately human scaled, with all the abstractions and operational rules one would expect to find in a domain similar the world we all live in, with one important difference. The difference is *magic*, a force which twists the most straightforward of simulation design into a hairy tangle, if not carefully managed. For example, the logic of an ordinary sword is fairly straightforward, but our designers would like that sword to acquire special behavior if it is enchanted by a wizard, say to do extract damage against certain types of undead creatures. Behavioral adjustments like this need to be able to happen often, and at runtime, with no code changes or server downtime. Python serves this requirement particularly well.

UO2 represents entities active in the virtual world using the SimObject pattern [2]. High-level SimObjects implemented in Python represent the logical simulation domain of the Ultima fictional universe. SimObjects represent both instances of characters, houses, and weapons, and encode the behavior and interaction of such objects. Associated Body instances implemented in C++ represent the physical aspect of the simulated objects. Body instances have mass and dimensionality, collide with each other, travel through space, animate in the world, and react to physical forces such as gravity.

UO2 defines several simulation subclasses, such as *Character*, *Armor*, and *Weapon*, and so on, which establish the base behavior for such objects in an “Earth-normal” environment. Operational simulation logic defined by the *SimObject* subclasses is split into two phases, a *test phase* and *execution phase*. All tests defined in the test phase must pass before any changes to the simulation are allowed to happen in the execution phase.

Base behavior of the *SimObject* subclasses is extended by an approach that combines inheritance and delegation via a mechanism we call “Properties”. Properties are additional pieces of behavior implemented as unbound Python functions grouped together in modules. Known interfaces on the base classes are established as *protocol sets* which can be extended in a chained manner by Properties. This approach allows both the test and execution phases of particular *SimObject* actions to be augmented by layering additional Properties onto the instance. Attaching a Property to an instance at runtime has the effect of changing the behavior of the instance as if a new subclass had somehow been inserted into the inheritance hierarchy *for that instance only*. Properties attached at the class level have the effect of augmenting behavior for all existing and future instances of that class.

As a simplified example, a Property might extend the *CanUse* and *Use* interfaces to perform additional checks on the testing phase and execution phase of the use of an Item. Given an ordinary instance of a Sword, a subclass of the *Weapon* class which is in turn a subclass of *Item*, a character can pick it up, wield it, swing it around, and so on. But for our example, a wizard has enchanted this sword so that it is magically more powerful against undead creatures, such as zombies, vampires, and skeleton warriors. As a side effect, the enchanted sword can now only be wielded by those characters who are pure of heart, those who have performed good deeds and have never stolen or cheated at dice. On the server, this results in the attachment of two new Properties, *UndeadSlaying* and *PureHeart*.

Attaching a Property to an instance triggers the processing of the list of methods which that Property augments, causing a reference to each unbound function of the Property to be acquired and stored in a collection maintained by the instance. In the example, *CanUse* and *Use* are known augmentation hook points, and *UndeadSlaying* defines a *Use* augmentation function while *PureHeart* defines a *CanUse* function, as shown in Figure 2.

When a character attempts to swing the sword, *CanUse* is called on the instance, which in turn calls all *CanUse* augmentation functions in its internal property collection. In this case, *PureHeart* performs an additional check during this test phase to determine if the character’s karma is sufficiently good. In the case that a

character's karma is sufficiently bad, the PureHeart *CanUse* return false and immediately breaks out of the calling chain. No *Use* methods are called because all tests must pass before any simulation execution happens. However, in the case that the character's karma is good, the PureHeart *CanUse* method return true and calling continues down the *CanUse* chain.

Assuming all other *CanUse* tests pass, the code goes on to call *Use*, which performs the usual work of the sword, and then calls all of the property execution functions. This includes the *Use* function defined by the UndeadSlaying Property, which performs a check to determine if the target is undead, and if so, to increase the amount of damage done to the target.

3. Client Interpretation, Server Arbitration

The Python portions of the UO2 architecture ride on an advanced third-generation design for threaded servers, message passing, network communication, and sophisticated 3D animation and rendering. Python interpreters run on both clients and servers.

On the server-side, the simulated world is divided up into arbitrary regions, each region running on one AreaServer process, as shown in Figure 1. SimObjects in the world may cross server boundaries at will, either by teleportation from one location to any other or simply by "walking" from one region to another across the boundary. Server boundaries can be adjusted to balance simulation load across AreaServer processes, and crossing server boundaries is completely transparent to the client.

A user's primary experience of the virtual world is that presented by the client. The UO2 client is a sophisticated engine for graphics and sound, enabling highly detailed and customizable avatars, special effects, foley, and soundtracks, which adapt to dramatic action in the world. The client process runs locally on the user's computer, generating real-time 3D graphics, text, and sound representing the state of the simulation. The client also accepts input from the user and translating that input into requests for the user's character avatar to perform actions on the server.

Once connected to the server, the client exchanges information in an encrypted stream, receiving simulation updates and sending action requests. Interaction at the logical simulation level is in the form of remote method invocations between Python SimObject instances and service Singletons on the server and client using a custom Python mechanism similar to Java's RMI [9] or Pyro [4]. Interactions at the physical simulation level are handled through specific C++ message classes for high speed processing of position and orientation.

An important design decision made early on in the life of this project was that the server would never trust the client, meaning that all simulation decisions are made by the server. In UO2, SimObjects running on the servers represent the “One True” version of reality. SimObjects running on clients represent proxies of SimObjects on the server, and are subject to correction and adjustment based on server arbitration.

This puts the client in the position of making requests for the server to perform if and only if all checks for that request pass on the server. In practical terms, the split between client request and server arbitration helps to maintain a consistent view of simulated reality across all clients. A user who attempts to modify the state of the shared simulation by hacking their own client will, at best, merely invalidate their local representation of the simulation.

4. Data Driven Classes and Properties

A primary strength of the system is the degree to which the simulation is data-driven. The AreaServers communicate with a DBServer, which presents a relational database to the simulation. An encompassing set of tables, views, and stored procedures, which we refer to as the *Master Game Database*, captures the data for the archetypes of our fictional world. The Master Game Database allows world builders and designers to significantly expand the world with new entities and behaviors simply by entering new data and without changing code.

The DBServer loads a handful of startup tables that define other tables to load and how to deal with them. Python code running on the AreaServer processes the startup tables differently depending on the kind of table. Some tables can be simply converted into symbolic “constant modules” in the Python runtime, which are imported and referenced by code or by other tables. Examples of these modules include *ParserCommand*, *PlayerMessage*, and *EffectContext*. Other more complex tables are loaded for additional processing and association as runtime data, such as *PublishedMesh*, *CombatStroke*, and *StartingCity*. Of course, tables can be both, such as *GameAttribute*, *Race*, and *EquipmentSlot*, which require additional processing but still present convenient constants that other code can use.

The most complex of the tables loaded and processed by AreaServers are those which define new subclasses of the standard base classes. Tables exist for each of the “leaf core classes” of the standard inheritance hierarchy, with additional rows for each subclass of the base class. Each row defines values for class attributes and a set of 0..n Properties which can be attached to the class. On AreaServer start, the contents of these tables are loaded and new subclasses are constructed at

runtime from this data. Properties associated with classes at this level enable all instances of that class to gain the additional test and execution behavior. This powerful feature allows designers to specify all variations of the core classes simply by changing data and mixing and matching Properties.

Since the set of SimObject instances on all servers constitutes the entire state of the virtual world, it is necessary to archive SimObjects to disk over time. The archival preserves an updated state of the world and enables the restoration of world state in the case of server shutdown. The DBServer again comes into play here, consisting of a set of tables for storing SimObject state, which we refer to as the Runtime Game Database. AreaServers periodically serialize SimObject instances using a modified version of the standard Python cPickle [8] and transmit the serialized data to the DBServer. New instances are serialized immediately upon creation and added via SQL INSERTs. When serialized again, instance data is archived using SQL UPDATEs. The system is also flexible enough to handle changing server boundaries, so that SimObject instances are automatically reloaded into the appropriate AreaServer region encompassed by the new boundary settings.

6. Future Work

The UO2 system will continue to be expanded, forming the basis of a reusable shared software architecture for future Origin Systems products. The flexible nature of the UO2 system allows for easy experimentation, allowing rapid expansion and adjustment. User feedback will drive the incorporation of new features and the general improvement of the virtual world experience as we go forward. Periodic major additions will be released to the user base over time, including the capability for users to own houses in the virtual world, tame and train virtual pets, and develop new professions within the game.

In addition to entertainment purposes, technology like the UO2 system has enormous potential for educational use [3]. Other advanced forms of computer interaction, such as wearable computing [6] will require new metaphors and interaction models, some of which may be perfectly suited for networked virtual world presentation.

I believe that the future of the Internet and the personal computing experience in general will be highly interactive, rich, and immersive three-dimensional persistent virtual simulations. Languages like Python and the development communities that grow up around it are a critical part of this future.

Acknowledgements

I want to express special thanks to everyone on the UO2 team at Origin, it's been a wonderful experience to work with such a talented, dedicated, and selfless group of professionals. Thanks to my friends and reviewers, Michael Crawford, Heather Dority, John Edwards, Edward Robinson, and David Stidolph. Also special thanks to the Python community members who have given of their personal time, David Ascher, David Beazley, Paul Dubois, Mark Hammond, and Mark Lutz.

References

- [1] Addams, S., *The Official Book of Ultima*, Compute Books, 1992.
- [2] Asbahr, J., "Beyond: A Portable Virtual World Simulation Framework", Proceedings of the 7th International Python Conference, Houston, Texas, November 10-13, 1998.
- [3] Bruckman, A., "Community Support for Constructionist Learning", ACM Conference on Computer Supported Cooperative Work, Seattle, Washington, November 14-18, 1998.
- [4] de Jong, I., Pyro Home Page, <http://www.xs4all.nl/~irmen/ap/pyro.html>.
- [5] Laurel, B., "Post-Virtual Reality: After the Hype is Over", *Computers as Theatre*, Addison-Wesley, 1993, pp. 199-214.
- [6] Picard, R., "Affective Wearables", *Affective Computing*, MIT Press, 1997, pp. 227-246.
- [7] Pesce, M., *VRML – Browsing and Building Cyberspace*, New Riders Publishing, Indianapolis, Indiana, 1995.
- [8] PythonLabs, cPickle – An Alternate Implementation of Pickle, <http://www.python.org/doc/current/lib/module-cPickle.html>.
- [9] Sun Microsystems, Inc., Java Remote Method Invocation (RMI) Home Page, <http://java.sun.com/products/jdk/rmi/index.html>.

Figures

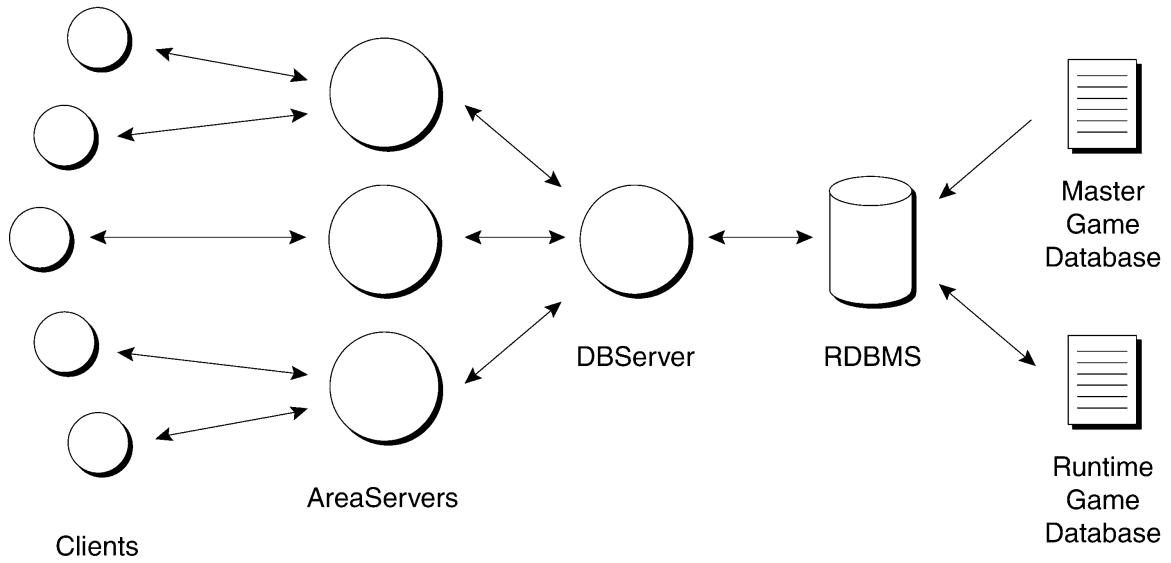


Figure 1: Simplified UO2 Data Flow

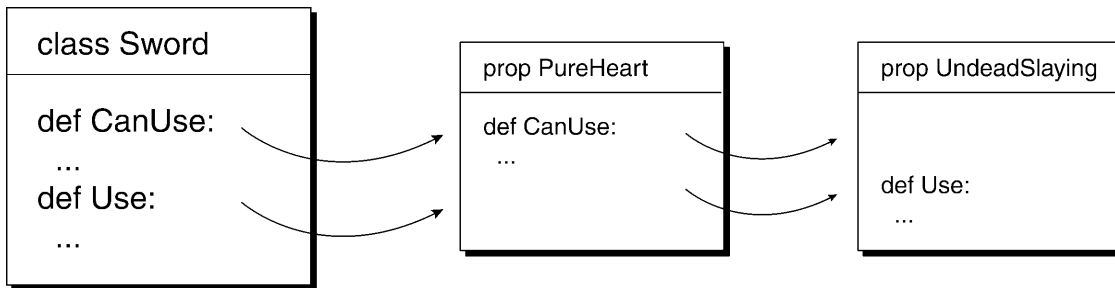


Figure 2: Chained Property Example