Overview

This document specifies the general architecture of the SkyNet Model that will be XML-based. In specific, how entities exist and interact and how data is stored will be explained and illustrated in detail.

Introduction

The current SkyNet Model consists of a hierarchical system where the world consists solely of entities. Each entity has a collection of attributes, a collection of behaviors, a collection of points (represented by an IArea) that describe its “physical” location and a collection of ProxyEntities that may play a part in certain behaviors. Entities may be parents to other entities (e.g. A tank may have a machine gun and a turret as its child entity).

Behaviors are executed primarily by using the attributes (or any other criteria that the behavior deems necessary) and computing a GameEvent to be broadcasted to the network.

The Idea of an XML-based model

Based on the above description, the whole SkyNet Model can be serialized into an XML file with a schema that is implemented right now as a compatible to an ADO.NET DataSet. The diagram of the tables is shown below:

(The diamonds represent the relationship; all are one-to-many relationship)
Using the tools provided by Visual Studio.NET, the XML Schema can be generated into classes that reflect the data (i.e. there will be a class called SkyNetModel which inherits a DataSet and can accept an XML file with the valid schema and data to be serialized into it).

The keys in the DataSet provide methods in which other objects can use to interact with the model. For example, a behavior can get all the attributes of an entity by calling SkyNetModel.Entities.FindByEntityGuid(some guid).GetAttributesTable().

The relationships in the DataSet provide constraints in which the data can be validated during addition, insertion and deletion of data. For example, adding an attribute with a certain EntityGuid will automatically require the existence that entity in the Entities Table. Also, updates and deletion will be propagated to the child tables as well.

**The Benefits of using an XML (DataSet) Data Structure**

- Data can be easily serialized into .xml files or transmitted through the network. The client model can simply call SkyNetModel.Merge(SkyNet Model from Server) and its data will be automatically refreshed. DataSet can also be written to the hard drive for debugging purposes.
- ADO.NET provides transactions inside DataSet and optimistic/pessimistic locking that ensures the ACID properties of changes that occur in the model. The original design of the model requires locking the entity when updating the attributes' Hashtable thereby restricting access to the entity from other objects. Using the new system, ADO.NET locks only the rows that are being updated and determines if phantom data or race condition can occur if concurrent access is allowed. This can allow other objects to access the model if they simply want to read but not change the data.
- Events are generated automatically by ADO.NET when rows are changed (added, deleted or updated). The client model can then easily pass these events into the View and would therefore eliminate the need to determine when and what to send to the View when messages are received from the network.
- When a server joins the network, other servers can serialize their model and send it over the network to ensure that they have the current model in memory. The original idea of passing each entity individually over the network is eliminated since only one single stream of data is required to recreate the entire model. Similarly, when clients request for updates, they will not be receiving potentially thousands of EntityUpdateMessages (as it is currently called) but rather an XML file that describe all that exist in a certain envelope.
- Since a DataSet can also be stored into a database such as a SQL Server or MSDE. The model need not only exist in memory. If we ever decide that we would want to stop running the server but retain the data for future execution, we can simply store the DataSet into a SQL Server for future execution.
Using SQL Server for Back-end Storage

The current direction that we are heading for in designing the Server Model is to use ADO.NET to represent the model in memory as well as to store the data in SQL along the way. Storing the data in SQL allows us to have a physical copy of the data in permanent storage with much faster storage and retrieval time than a file that exist on a hard drive. We are currently using MSDE, the embedded version of SQL Server, to host the data. There is much more flexible licensing requirement on MSDE than SQL Server and if ever we decide to scale up to a real SQL Server (possibly a cluster), we could easily replace MSDE with SQL Server. (MSDE is downloadable from Microsoft)

Another benefit of using SQL as the backend is that more complex queries, if needed, can be executed directly on the SQL Server instead of looking at the DataSet model in memory. Of course, the results would be fed back into the DataSet and from a “user” standpoint, they will not notice whether the model is using the DataSet in memory to retrieve the results or whether the SQL backbone is used. Examples include the serialization of the model to the client with a certain envelope. When the client request for all entities within a certain window, while it is possible to just look at the DataSet in memory and grab only those that is within that window, it would have been much faster if we can query the SQL database with a query that only returns entities (with their related children) within that envelope and serialize the results instead.

Another advantage of using SQL as the backend is the support for SQL in .NET Enterprise Services. In SkyNet, when events happen across server boundary, the original idea is to use an n-way handshake system where servers would agree on processing a certain event and execute them at the same time. This requires multiple messages to be passed around when events occur. .NET Enterprise service provides transaction support between different computers using Microsoft Distributed Transaction Coordinator (included in Windows 2000 onwards, formerly known as MTS). It uses a two-phase commit system such that transactions either occur or they don’t occur across different computers.

We are actively trying to harness the available tools from Enterprise Services so that we will not be required to deal with complex locking and handshake issues when inter-server communication is required. The Server Model can be hosted as a Windows service if necessary and more advanced tools can be used if performance and scaling is of a concern. Object pooling and multi-threaded processing can be enabled for Enterprise Service objects (a.k.a components) and if we ever want to host SkyNet as a Server-Client system (instead of p2p) we can use Microsoft’s Application Center to host the components on a server farm.

While it might be possible that we can write our own transaction system for inter-server communication, it is probably better, given the current time constraint and other human-factors, to use readily available classes from .NET to implement our ideas.