

A Data-Driven Game Object System

By Scott Bilas

Abstract

In a typical game, nearly all dynamic, interactive content is managed by “game objects”, which must perform complex tasks like animating, speaking, pathfinding, persisting, triggering, and networking. The textbook method to designing this system is to construct a hierarchy using the C++ type system, yet designers want their objects to perform as many interesting and varied tasks as possible. As engineers we must then struggle to make their desires fit into the constraints of our strict type system, recompiling over and over again to keep up with even slight changes in the design (not to mention keeping the level editor up to date).

In this talk we will cover a better way of building and managing game objects. Throw away the hierarchy, break the tasks down into C++ and script components, and assemble the components into objects. Data-drive the assembly process and now you have a powerful and quick system for building new types that requires no engineering time to manage! The design given in this talk is the same as that used in Dungeon Siege, which manages hundreds of unique object types.

\$\$\$ more on this

The Problem

Note the application of this design (content-heavy role playing game, 4000 total object types, 70,000 objects in the smaller single-player game world, continuous streaming engine, etc.).

Talk about sample types of game objects (monster, boss, tree, command, waypoint).

\$\$\$

The Textbook Solution

The game object design problem is, to a large extent, about polymorphism. In order to minimize engineering work, we create systems that try to maximize the similarities among our game object types.

Introductory textbooks to C++ usually have a section in them on polymorphism. They describe breaking down a problem in an object-oriented way, and invariably give the classic employer/employee example. In this example, the employer

[diagram]

Implementation

Next cover the system used in Dungeon Siege:

- Explain the need for a general-purpose hierarchical data language such as XML (give examples from Dungeon Siege's GAS language). Also need a simple query system for this database.
- Detail the organization and classes used in the system – hierarchical data configuration with instancing and override support, the component-based game object system that relies on it, and the object databases that manage all of this. Answer the question, “what is a component?”
- Show how to build components out of scripts and seamlessly integrate them into the system so that there is no apparent difference between C++ and script components to the rest of the system.
- Finally, detail how the level editor fits into all of this. The editor must save/load object instances for a level and be able to override properties of components, keeping up to date with changes in those components. Show how to gain this functionality nearly for free.

Advantages

- Easy to use with a fast turnaround. Designers can very quickly have their ideas prototyped, usually without engineering intervention. They can create new types and easily insert them into the game. New functionality can be added by a scripter, with engineers only providing occasional new function support.
- Memory efficient. As components are groups of related functionality, it is easy to remove components from types that do not need them. Better yet, in a multiplayer, server-only components can simply not be loaded on clients.
- Self-organizing. The types are organized based on required functionality, not programmer convenience, and so naturally organizes a game's object types into an easy to understand specialization tree.
- Self-documenting. One of the problems of engineering is documenting and maintaining the schema and rules of the engine – this system makes it easy to see how objects may be constructed at a glance by encoding the rules and constraints into data that the designers can see and even tune.

Disadvantages

Any complex system comes with disadvantages. These may be avoided by actively managing the design and implementation of the components.

- More prone to spaghetti because components very often need to talk to each other, and so become interdependent. Various operations may become order-dependent on the components. Assigning each component a priority can help.
- Must be careful about how the data templates are structured to make sure that components aren't added arbitrarily (increasing memory usage and CPU load). Should regularly evaluate the components to make sure they are slim, though the necessity of this completely depends on the game being built.

References

Talk

And that wraps up the dynamic part of this system. Pretty simple stuff, the more complicated part is the code for static content.

Schema: GoDataComponent, GoDataTemplate, ContentDb, components.gas.

Data: templates.gas, etc.

Implementation: Go, GoComponent, GoSkritComponent, GoDb.

Usage next. When you want to build a new type of Go, you pick and choose which components it needs and add them into the data specification. If you're

Copied Unreal for script -> editor thing.