Example Game Architecture and Analysis

By Jeff Plummer

Jeff.plummer@jnyinvestments.com

I am writing this article because I have heard many people ask, "how should I design my game's architecture?" I think it would be a helpful exercise to post up a sample game architecture, and discuss the pros and cons, such that people can get a feel how a game engine might be designed. It is my hope that people will continue the analysis of the architecture I have posted, as well as post up other game architectures for analysis and discussion.

Introduction to The Delta3D Engine (http://www.delta3d.org/)

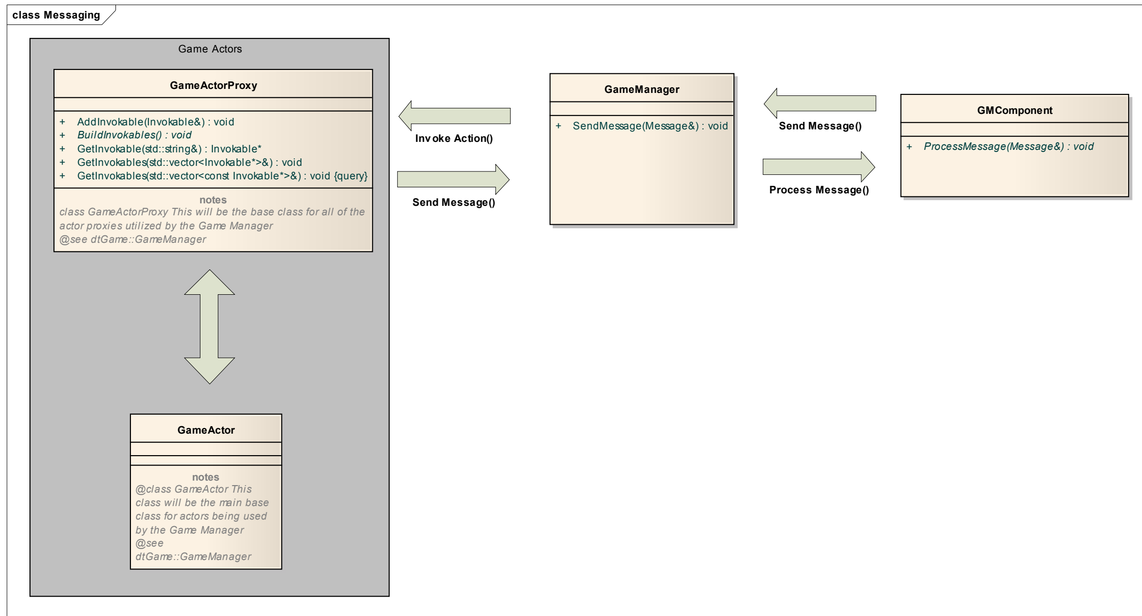
The Delta3D Engine is an open source complete 3D game engine put together by the naval postgraduate school. It is a pretty well designed and documented game engine that has a few very interesting approaches that should make for an interesting discussion.

Disclaimer: Analysis and diagrams were created by me, and heavily influenced by their excellent documentation. I chose not to use their design document diagrams directly because I wanted to change things around a bit so I could create a much shorter design description. I have had no role in the development of this engine. I've just played around with it a bit, and so any errors can be blamed on me.

Architecture Description

In this section we will begin by discussing the high level architecture, and then delving in a little deeper into each of the major subsystems.

The 10,000 Foot View:

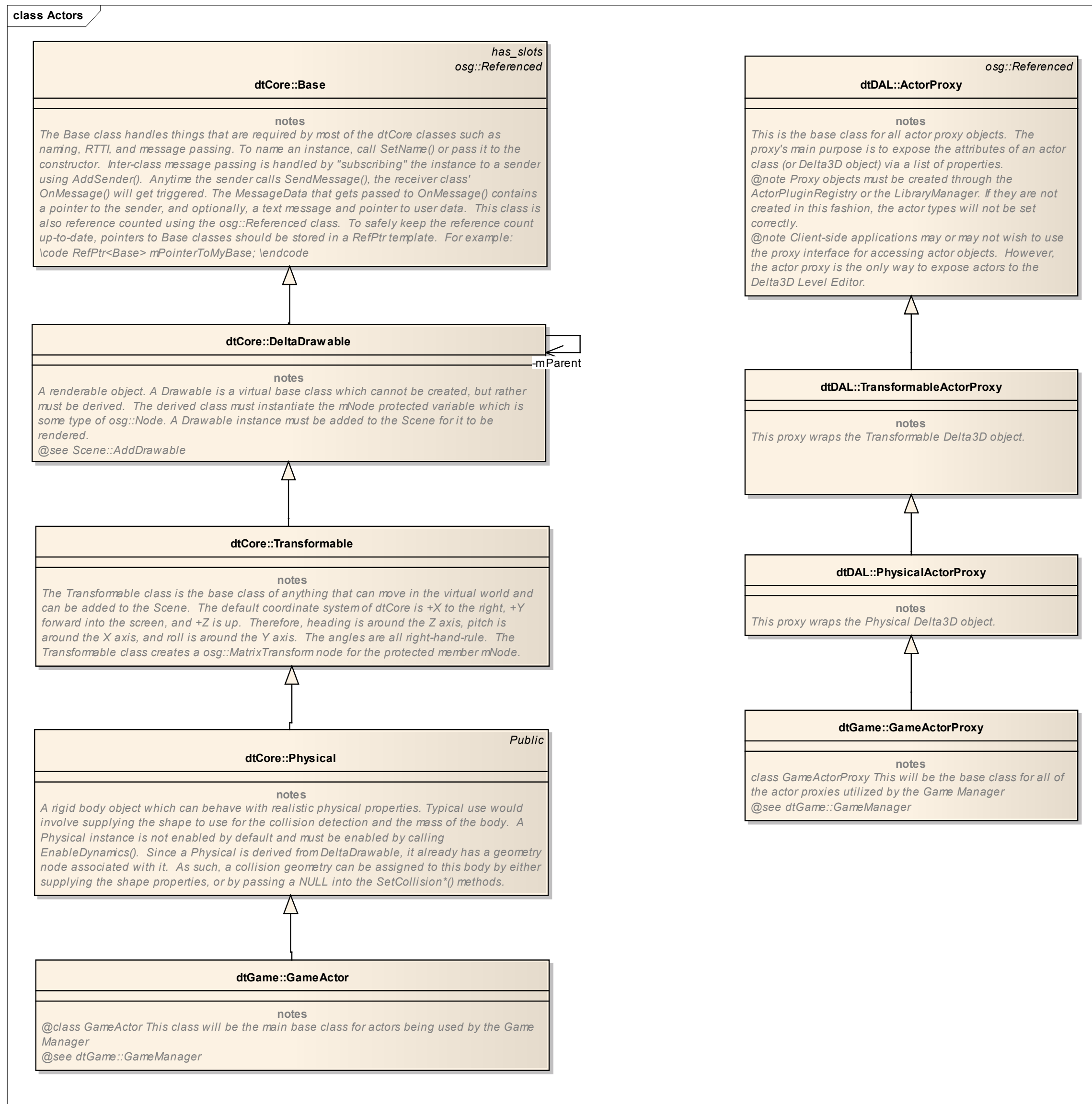


At the highest level, the Delta3D engine is very simple consisting of only 3 main subsystems. At the heart lies the "GameManager", which is responsible for managing actors and message passing. The GameManager will act as the communications conduit between the other two major subsystems.

The second subsystem in the engine is the game actors. Game actors are the entities that exist in the simulated world, and are further divided up into two sub-pieces: the proxy and the actor. The actor is the real object, and may contain data and logic. The proxy exists to make access to the networked actors transparent, and provides a generic data access interface for the game manager and other tools (we'll discuss this later).

The third and final subsystem is the "Components". Components are agents that receive messages and perform arbitrary logic in response to those messages. They could perform logic like sending data over a network, or updating a player's points due to scoring a touchdown in a football game.

Actors:



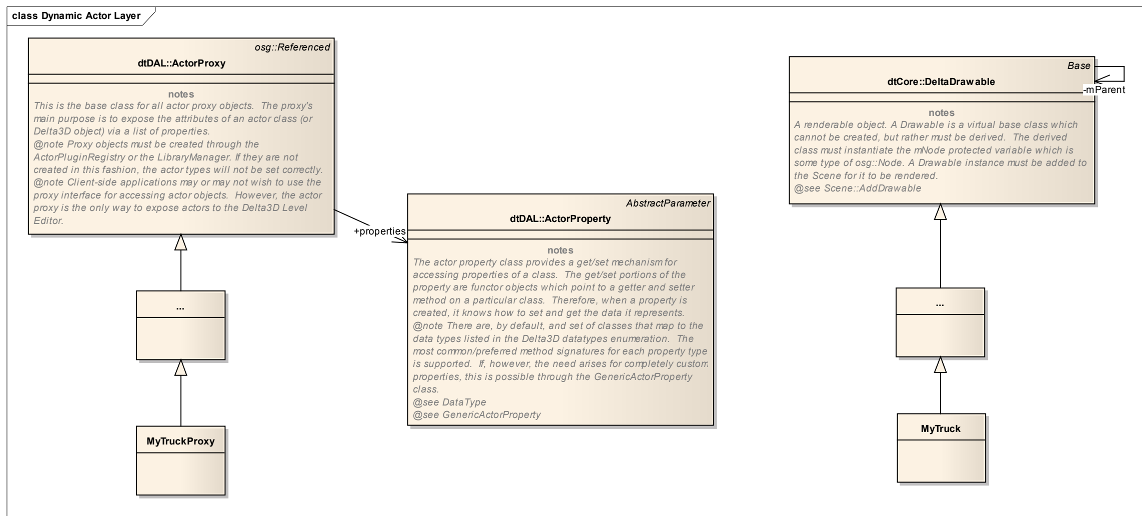
Actors are the entities in the simulated world. Actors and their proxies have (some of) their functionality implemented in a hierarchical fashion. "DeltaDrawable" provides capabilities draw the actor using the Open Scene Graph graphics technology (www.openscenegraph.org). You'll also see "Physical" which provides capabilities for the actor to have rigid body physics using ODE technology (www.ode.org). By extending your actors from the GameActor class your actor will already have these capabilities.

Dynamic Actor Layer (More about actors):

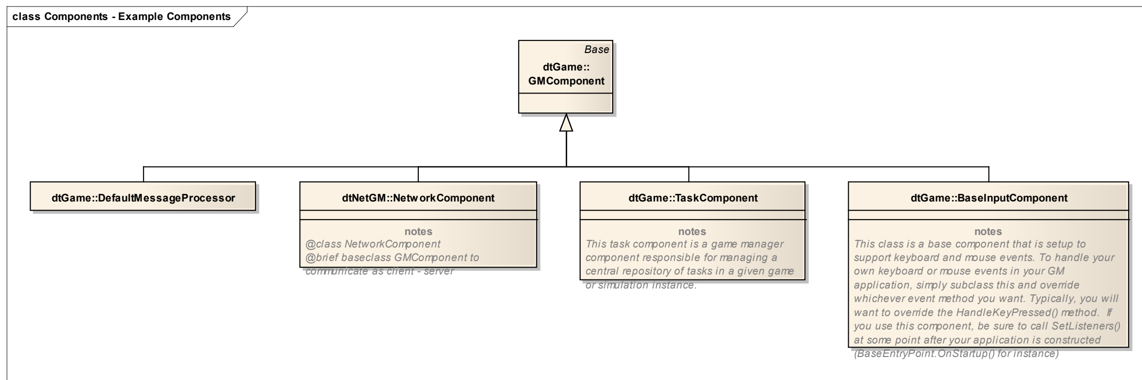
The dynamic actor layer exists to provide generic access to arbitrary actor objects allowing tools and editors the ability to interface with any kind of actor. For example, a level editor tool should be able to allow the user to manipulate any game actor object, regardless of the actor object. To do this, the Delta3D engine extends the notion of an

"actor proxy" to create a translation layer between actor data, and a data format that the engine and tools can understand.

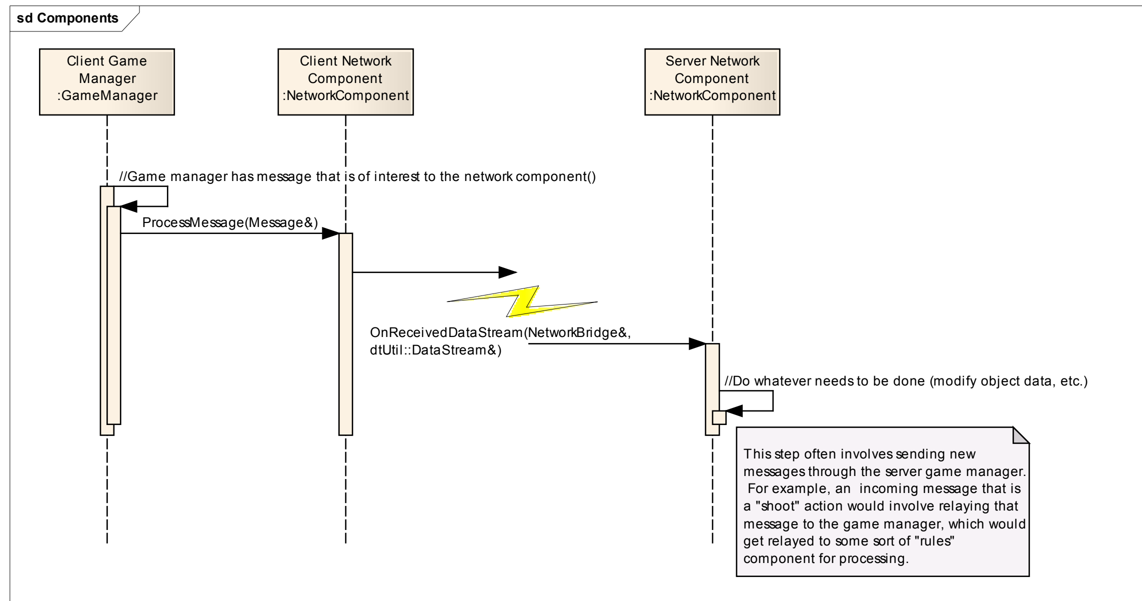
The design is extremely simple. Actor creators must create two objects for every actor entity they want to create. The developer must create the actor itself, and a proxy object that maps named "properties" to the actual actor object data. Tools can then be designed to work with generic "properties", and the result is the arbitrary object is manipulated as well.



Components:



The "Components" subsystem consists of message processors that provide a very powerful and flexible way to add capabilities to the game engine. Above you can see a few example components. For example, the "BaseInputComponent" provides the capability to process mouse clicks and keyboard presses, and a game developer could extend this class to add game logic like shooting when the space bar is pressed. Networking capabilities are also implemented using a component, and you can see a simple execution in the diagram below.



Components are neat because ANY logic could be written in a component. Components just listen for messages, and perform logic. Game developers could use components to implement game logic like rules, or technology capabilities like networking. Components provide a great extension point in the engine to create game specific behaviors.

Architecture Analysis

In this section I am going to describe how the architecture relates to quality attributes of the game engine.

Performance: I didn't see anything that would have a negative impact on performance on a single core processor as compared to other engines I have used. I do however see potential issues when you discuss performance on multi-core processors. With some technologies tied into the actor class hierarchy there aren't very good ways to separate the functionality into different processors. "Component" functionality will be much easier to separate across CPUs. At present however, there is no notion of component dependencies (that I could see, anyways), which could make optimal component CPU allocations difficult. (e.g. You really should process the "rules component" before you execute the "networking component").

Modifiability: I really like the "component" capabilities. They allow for a lot of flexibility. However, for some reason they chose to integrate physics and graphics into the actor hierarchy. That means if you ever want to move away from open scene graph, ODE physics, or even expand these capabilities you must deal with the guts of the engine. If these had been implemented as "components" then presumably I could swap in new graphics engines "components", etc. without breaking anything.

Portability: There aren't any architectural constructs that I saw specifically designed to ease portability. However, the engine uses highly portable libraries, like open scene graph, so the resulting engine is fairly portable.

Reusability: "Components" and to some degree actors are reusable. "Components" are great because if you put in the effort, you can design smaller, generic components that when combined result in your desired functionality rather than large game specific pieces of code. Actors as a whole are reusable, but there is no way to use pieces of an actor other than cut-n-paste code from the actor or extending the hierarchy – which can lead to issues when you discover your hierarchical tree doesn't allow you to create objects that have the capabilities of an arbitrary subset of those base classes (see rationale for the "decorator" design pattern for more info). So if you wrote some great AI code inside of an actor, they only way to add that AI to other actors will be to cut and paste.

Time to Market (or how fast can you get it done): Delta3D is pretty good with this quality because it is so easy to use and already provides a great deal of the functionality every game needs. As a developer, you primarily need to develop actors and "components", and since these are conceptually easy to understand, and physically easy to develop, work can be done fairly quickly.

Cost: It's free.

Conceptual Integrity: I'm going to vote low on this quality. The engine has very simple concepts to use, but it provides mixed messages as to how to extend the engine. For example, how should I add character behavior AI for my game? Graphics and physics capabilities are added by extending the actor object. Networking and rules are implemented by creating "components". My personal opinion is they should pull out all the functionality from the actor hierarchy and implement it as "components". Unfortunately, I am not emperor of the world – so no one asked me ☺.

In Closing

I hope people found this analysis interesting. Software architecture in games and simulations is a fascinating topic, and I hope this article will inspire genius in the game architects that read this.

Questions / Comments? Email jeff.plummer@jnyinvestments.com

Creative Commons Copyright © Jeff Plummer.
Share it, change it, re-write it. Just remember to mention what a swell guy I am.