

A Software Architecture for Games

Michael Doherty

Department of Computer Science

mdoherty@uop.edu

Abstract

This paper describes a general architecture for real-time game software. The architecture is designed to maximize reusability in both single player and networked multiplayer modes. The architecture separates generic components from game-specific components by positioning the object system to minimize coupling. As a result, the architecture is easily adapted to a client-server configuration for multiplayer modes.

In addition to presenting the general architecture, that architecture is used as a framework in which to discuss several design options and the tradeoffs between them. In particular, we will discuss how the choice between an object-centric and a component-centric object system affects the design of other components of the game software. In addition, issues related to the synchronization of object systems for networked multiplayer modes are discussed.

Introduction

Video games are becoming increasingly sophisticated and the software programs that drive them are becoming correspondingly more complex. As a result, the programming teams that build game software are becoming larger and the roles of individuals on those teams are becoming more specialized. In addition, the game industry has grown and matured to the point that the expectations of the industry are becoming similar to the expectations in other areas of software development. Thus, game programming professionals are becoming increasingly aware of the need for better development methodologies and software engineering techniques. Discussions in workshops at recent Game Developer's Conferences [LS02, LI03] are evidence of this.

While there is considerable literature on various specific topics in game software, most notably graphics [Eb01, EGL01] and physics [Bo02, Eb03], very little has been published on the higher level design and engineering of game

software. Rudy Rucker's new book [Ru02] is one of the first to attempt to specifically address software engineering for games. However, even this book fails to give a clear and general high-level architecture.

Another attempt to address game architecture is Rollings and Morris [RM00]. While they do identify the principle software modules for game software, they do not address how those modules interact to provide the functionality required of the system or how the architecture can be adapted for multiplayer configurations.

To address the need for a better understanding of architecture and engineering in game development, we have developed a general architecture that can be used as a starting point for the development of new games. A key feature of this architecture is the use of a central object system [Ch02, Du03] to minimize coupling between elements. This has the advantage of clearly separating game specific components from the more generic components and of minimizing the impact of incorporating networked multiplayer game modes.

While we do not claim that a software architecture of this nature has never been used, we are not aware of any public documentation of such an architecture. In particular, architectures that specifically incorporate the object system as a central component have not been addressed in available literature. We hope that this document will have value in helping to form a shared vocabulary for game software design, and that it will serve as a vehicle for identifying areas for future research.

Top-Level Components

The architecture has the high-level structure shown in Figure 1. The functional components, the game engine, the simulation and the data manager all interact with the object system. The game engine is responsible for presenting the game to the player and for receiving player inputs. This includes the generation of graphics, audio and any other feedback to the player. The

simulation is responsible for updating the state of the virtual game world in response to player inputs and the rules of the game and virtual world. The data manager is responsible for retrieving game data from the file system or some other persistent storage and for managing storage and retrieval of game state for save/load game functionality.

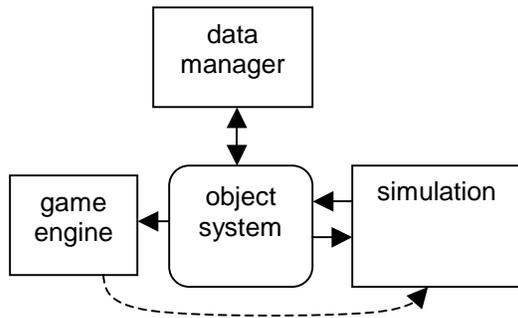


Figure 1: Top-level Components.

The object system is responsible for maintaining the state information describing all objects in the game world. This system resides in memory for real-time access to all game state information.

At first glance, this architecture appears to be a fall back to functional decomposition, and as such may be disagreeable to readers who are more inclined to think in terms of object-oriented designs. However, there are a number of advantages to this approach, and, as will be shown below, it does not preclude a more object-oriented design for more detailed levels.

An initial advantage of this approach is that it minimizes the need (and the inclination) to specifically define the data flowing between modules. In initial attempts to develop this architecture we found that overly specifying these data flows was impractical and unproductive. Additionally, attempting to stay true to an object-oriented perspective resulted in bloated messages between modules, which essentially amounted to "pass a large amount of information".

Adding the object system as the central interface between the top-level components allows the designer to defer the identification of specific types of data (much of which is game specific) to a more appropriate time in the design process.

Separating the generic components (the engine) from the game specific components (the

simulation) was key to making the architecture more general. Also recognizing that the engine component has read-only access to the object system later influenced the networking specification.

Notice that there is one data path between the game engine and the simulation that is not through the object system. This path is for passing player input information from the game engine to the simulation. The primary reason for identifying this data path separately from the object store is that it limits the interaction between the game engine and the object system to a read-only interface. This has important implications in simplifying the processing that will have to be done by the networking layer in multiplayer modes.

The Game Engine

The game engine component is responsible for interacting with the player, through the computer hardware and operating system. In most cases, the game engine can be designed and developed as generic software for a particular game genre.

In Figure 2, the primary modules in the game engine are identified. This is not necessarily a complete list of modules, but is sufficient for typically supported hardware. For example, a tactile module might be added to support force-feedback devices.

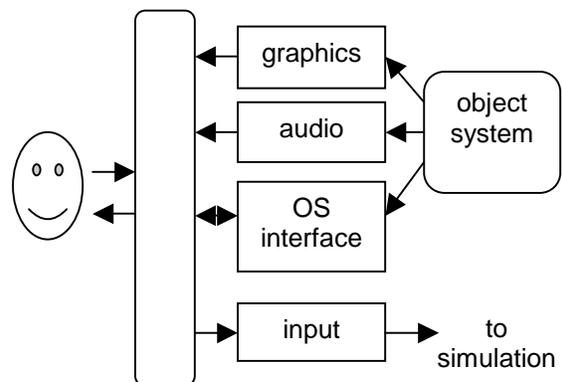


Figure 2: The Game Engine.

The vertical bar in this figure represents the player's computer, and the face represents the player. Note that the bar representing the computer is not strictly hardware, rather it includes the operating system, hardware drivers and support software such as DirectX [Ms03] or OpenGL [Op03].

One characteristic of the game engine modules is that their functional decomposition is natural and there is little or none coupling between them. In addition, these modules have very little internal state and the data flow through them is uni-directional.

While the internal structure and algorithmic design of these modules is non-trivial, they are better understood and documented than many other components in game software.

The Simulation

The simulation is the heart of the game itself. This component realizes the virtual world of the game and maintains the rules of the game. The components of the simulation module are shown in Figure 3. The simulation control module receives player input information from the game engine and passes it to the player AI module. It also passes control information, such as system time and elapsed time, to the other modules. The dashed lines represent this control information.

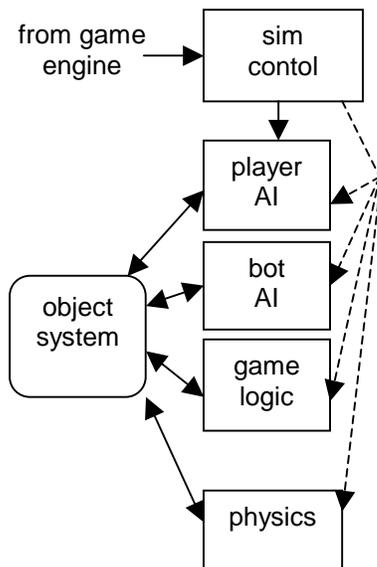


Figure 3: The Simulation.

The remaining modules encapsulate artificial intelligence (AI), physics and game logic. The physics module is concerned with the interaction of objects in the virtual world according to the physical properties of that world. The AI module is concerned with the internal thought and decision process within (virtually) animate objects. Finally, the game logic module handles those issues that are not specifically part of the virtual world in which the game is played, but

are rather computations and constraints inferred from the rules of the game.

These modules do not interact directly; rather they interact through the information that describes the state of the objects in the game world that is stored in the object system. In our discussions, we found it useful to think of the interaction between AI and physics as follows. AI generates impulses based on the internal state of animate objects. For example, the AI for an object might generate an impulse to move towards water or to swing a fist. Physics turns impulses into actions and ensures that the results of those actions do not violate the physical laws of the virtual world.

Networked Multiplayer Architecture

In a networked multiplayer game, a number of players interact within the same virtual environment, and that interaction is mediated by their individual computers. Thus it is natural that the simulation component of the game is processed on some common system (the server), while the rendering and interface of the game engine is processed on the individual systems of the players (the clients).

Since all communication between the simulation and the game engine is through the object system, the object system is the natural point from which to deal with networking. Each client system needs to have a copy of the object system that contains that is being updated by the simulation. Thus the networking problem reduces to the problem of synchronizing the object systems on the clients with the one on the server. This is shown in Figure 4.

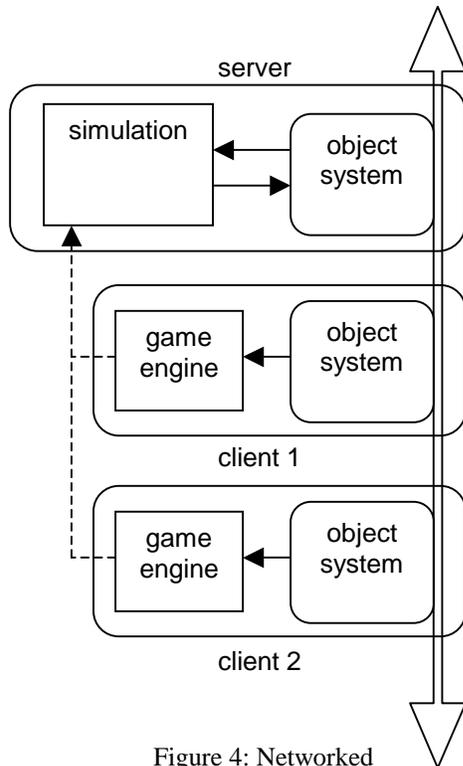


Figure 4: Networked Multiplayer Architecture.

Since the interaction between the game engine components and the object system is read-only, the problem of synchronizing the copies of the object system with the original one requires communication in one direction only, which significantly simplifies the problem.

Although the object system synchronization is unidirectional, there is information that must flow from the clients to the server, as shown by the dashed arrow in Figure 4. This is the same player input information that was identified in Figure 1. This information is typically very small (relative to the size of the object system) and is generally on the order of tens or hundreds of bytes. It is for this reason that it is useful to consider it as a separate networking problem.

In order to reduce network traffic only changes to the server's object system should be sent to clients. To accomplish this involves solving two problems: identifying the data that has been changed and merging the changed data into the client object systems. These problems have been solved in the context of database systems using deltas that capture database state changes. Depending on the structure of the object system, solutions for object-oriented state [DHR96] or

relational database states [HJ91, JH91] may be applicable.

Network traffic can also be reduced by recognizing that each client only needs to see a subset of the object system. Designing such a solution would require analysis of the trade-off between the reduced traffic and the computational expense of determining what data is required at each client.

A component that is not shown in Figure 4 is the data manager. As stated above, the data manager is responsible for moving data between the object system and persistent storage. While it is possible to implement the multi-player architecture with a single data manager interacting with the object system at the server, this would lead to unacceptable network traffic, particularly when loading large amounts of art resources such as graphical models or audio files. The solution is to put a reduced data manager at each client, which loads the initial state of the object system. Since multiplayer games do not typically have the capability of saving games, this solution is sufficient.

Object System Design

Because of its position as the central interface between all other components and its critical function as the keeper of all data, the design and implementation of the object system can have a strong impact on all other system components. It also impacts the flexibility with which game designer can make changes to the game.

The design space for the object system ranges from object-centric to component-centric storage [Ch02, Du03]. An object-centric system is based on the familiar object-oriented concept of encapsulation, in that all data relevant to a particular object is stored together as a unit. Data in such a system is accessed by first locating the object and by then locating the relevant data within that object. In a component-centric system,

In either case, the object system is a map structure: $\langle oid, attr \rangle \rightarrow value$, where *oid* is a unique object identifier and *attr* is an attribute identifier. The type of the stored value can have arbitrary complexity. Under object-centric storage, the object stored is viewed as having the behavior $oid \rightarrow attr \rightarrow value$, while under component centric storage the behavior is

viewed as *attr* -> *oid* -> *value*. While these two views are functionally equivalent, they have significant impact on the implementation efficiency and the ability to have flexibility in the game design process.

The trade-offs between object-centric storage and component centric storage are identified by Duran [Du03]. The object-centric benefits are familiar to proponents of object-oriented design:

- It is intuitive. The internal representation matches the way that people think about the world.
- It is easier to insert and extract objects from the virtual world.
- Objects can have associated code (methods, member functions).
- The scope of components can be controlled.

The benefits of component-based storage are:

- It can be more efficient. Each component can be stored in a data structure appropriate to its access patterns.
- It enforces a data-driven design philosophy. Since objects cannot be anything but data, all behavior must be defined in terms of data.

The last point is important, because it exemplifies an issue that is somewhat unique to game software. Due to the nature of the industry and its product, game design must generally overlap game software development. This is partly true due to the business climate that tends to push development into very aggressive schedules. More importantly, it is due to the fact that the game design will naturally evolve as prototypes are developed and tested.

This has led the game industry to favor data-driven architectures [Bi02]. This is also sometimes referred to as a soft architecture [RM00]. The advantage of a data-driven architecture is that game designers can modify the game without involving programmers. In an object-oriented architecture, where behavior and attributes are encoded into software objects, the code must be modified and recompiled in order to change even the most basic aspect of the game design. In the environment of game development it is obviously a benefit to allow game designers and game programmers to work in parallel, but independently.

Object System Implementation

An example of an object-oriented approach to object system design is the Tiger Game Kit (TGK) [Do03], which is a generic game engine for 3D real-time games developed at the University of the Pacific. The object system is defined as a set of virtual classes, which must be implemented by the game application. Although TGK only provides the game engine, applications built with TGK have naturally maintained a pure object-oriented approach.

Because of the nature of object-oriented design and implementation, it is not clear that the architecture described here is directly applicable, since the functionality ascribed to the various system modules (physics, AI, graphics, and so on) generally becomes distributed in the implementations of the relevant objects. The use of aspect-oriented programming [EFB01, Ao03] would be useful in bridging this conceptual gap. An aspect is a functional unit that crosses multiple system modules and object boundaries. If the object structure is considered to be the horizontal organization, then modules such as physics and AI can be incorporated as vertical layers that involve many objects.

Examples of object systems that are essentially component-centric are those of the Thief and Deus Ex games from Ion Storm [Ch02, Du03]. Because these games are developed in primarily in C++, which does not have support for the component-centric model, the developers of these games had to reinvent many common object-oriented concepts such as inheritance and delegation.

Since the object system is essentially an in-memory database with (possible) persistence, the it is useful to think of the implementation in terms of the data models common to database management systems. The techniques for object-oriented databases naturally apply to object-centric storage. The relational data model can be easily adapted to component-centric storage. Each component can be modeled as a relation (table), identified by the relation name. Object identifiers can then be used as primary keys into the component with values held in a second component attribute.

We are not suggesting that object systems should be built by incorporating a database management system. Rather, it seems that the theory and

implementation techniques developed in the field of databases can and should be reapplied here. The idea of using deltas to facilitate synchronization of object systems was mentioned above.

The techniques of database systems may also prove useful in the development of hybrid object/component based system in which the data can be viewed differently by different modules. Persistence models for Java, in which the data is mapped into a relational database, have shown that this can be effectively accomplished.

The need for networked object systems and persistent object systems results in a need for an object identification scheme that is not based on the pointers or references typically found in programming languages. In component-centric object systems, the need for non-pointer OIDs is obvious. This is another area that has been well researched in both databases and distributed systems and their solutions would most likely be applicable.

Future Work

There are two areas of future work and research. The first is further study and analysis to validate the architecture proposed here. The second is more fundamental research to further the state of the art in game software development.

Validation work:

While the approach to networking through replication of the object system is conceptually sound, more analysis is necessary to determine if it will lead to feasible implementations. There needs to be both a formal and an experimental analysis of the size of rate of change of typical object systems in order to determine if it is practical to run the simulation module entirely separate from client machines. If the resulting network traffic is too high, some of the simulation will have to be done on the client system, which will make synchronization of the object systems more difficult. One possible solution is the use of compensating of estimating simulations that fill-in data values that cannot be synchronized with the primary simulation in a timely manner.

A common part of many game systems that has not been addressed here is scripting languages.

A scripting language allows game designers to define object behavior in a language that is independent of the primary development language. Scripting languages are generally interpreted, which furthers the soft architecture approach since designers do not have to deal with compilation and build issues. Investigation of scripting languages and their incorporation into our general architecture is required.

Fundamental Research:

While a huge variety of object systems of different types have been developed in the game industry, very little formal analysis has been done. An analysis of these systems relative to the data models used in database systems, programming languages and distributed systems would help to further characterize these systems. This would in turn lead to the ability to utilize more formal techniques in the design and development of future object systems.

Possible areas for investigation of future object systems involve designing hybrid systems that allow different perspectives of the data to different software modules. The use of object wrappers, which provide apparent encapsulation over a component-based object system, is one possibility. The incorporation of aspect-oriented techniques is another.

The use of deltas [HJ91, JH91, DHR96] for change management has a number of potential applications. As mentioned above, delta technology could be used for identification and merging of state changes for synchronization of replicated or distributed object systems. Deltas could also be incorporated into the data manager for more efficient save and restore game features. There is also the potential to use deltas to merge saved game states, which could lead to interesting game play such as allowing a player to travel back in time and change previous actions.

A general purpose networking solution, based on the analysis of object systems should be developed. Such a solution should be independent of game specific details to improve reusability. Theoretical models could then be used to analyze the requirements of specific games and object system implementations.

Conclusion

We have presented a general purpose software architecture for single and multi-player games. While the architecture is not necessarily unique, the documentation of the architecture is a first step in the development of more rigorous investigations into the science of game software development. The architecture incorporates the perspective of a single object system that is unique to game systems, in a manner that facilitates the discussion and understanding of other system modules.

The architecture shows that it is possible to maximize reusability of system modules by cleanly separating generic modules from game specific modules. The use of the object system to facilitate this separation was central to the discussion. That separation also leads to clean migration from a single player architecture to a networked multi-player architecture.

We have identified a number of areas for potential future research, mostly involving analysis of existing object systems and development of new object systems. It is hoped that continued development and discussion of software architectures such as the one presented here will lead to additional research that will improve the development process and the quality of game software.

Acknowledgements

Significant portions of the architecture presented here were developed during discussions in my course on interactive virtual environments at the University of the Pacific during the spring of 2003. The work presented here is the result of the collaborative efforts of the following students: Elizabeth Basha, Nick Duncan, Jordan Geiman, James Han, Marina Lau, Marinna Lee, Dang Lu, Tam Ngo, James Robertson, Dan Spengler and John Worthington. Special thanks to Michael Stone for facilitating the group discussions and leading the group to their results.

References

[Ao03] "Aspect-Oriented Software Development Website". <http://aosd.net>. March 31, 2003.

[Bi02] Bilas, Scott. "A Data-Driven Game Object System". *Game Developers Conference Proceedings*, 2002.

[Bo02] Bourg, David M. *Physics for Game Developers*. O'Reilly, Sebastopol, CA, 2002.

[Ch02] Church, Doug. "Object Systems: Methods for Attaching Data to Objects and Connecting Behavior". *Game Developers Conference Proceedings*, 2002.

[DHR96] Doherty, Michael, Hull, Richard and Ruppawalla, Mohammed. "Structures for Manipulating Proposed Updates in Object-Oriented Databases". in *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. June 1996. 306-317.

[Do03] Doherty, Michael. "Tiger Game Kit Homepage". <http://bailey.cs.uop.edu/Doherty/tgk/home.htm>". April 13, 2003.

[Du03] Duran, Alex. "Building Object Systems: Features, Tradeoffs, and Pitfalls". *Game Developers Conference Proceedings*, 2003.

[Eb01] Eberly, David H. *3D Game Engine Design*. Morgan Kaufmann, San Francisco, CA, 2001.

[Eb03] Eberly, David H. *Game Physics*. Morgan Kaufmann, San Francisco, CA, 2003 (in preparation).

[EFB01] Elrad, Tzilla, Filman, Robert E., and Bader, Atef. "Aspect-Oriented Programming". *Communications of the ACM*, 44.10 (2001): 29-32.

[EGL01] Engel, Wolfgang F., Geva, Amir and Lamothe, Andre (Editor). *Beginning Direct3D Game Programming*. Premier Press, 2001.

[HJ91] Hull, Richard and Jacobs, Dean. "Language Constructs for Programming Active Databases". in *Proceedings of the International Conference on Very Large Databases*, 1991, 455-468.

[JH91] Jacobs, Dean and Hull, Richard, "Database Programming with Delayed Updates". in *International Workshop on Database Programming Languages*. Morgan-Kaufmann, Inc, San Mateo, CA, 1991.

[LS02] Llopis, Noel and Sharp, Brian. "By the Books, Solid Software Engineering for Games". <http://www.convexhull.com/sweng/GDC2002.html>. March 18, 2003.

[LI03] Llopis, Noel. "By the Books: Solid Software Engineering for Games". <http://www.convexhull.com/sweng/GDC2003.html>. April 8, 2003.

[Ms03] Microsoft Corporation. "The DirectX Homepage". <http://www.microsoft.com/windows/directx>. March 30, 2003.

[Op03] "OpenGL Website". <http://www.opengl.org>. March 30, 2003.

[RM00] Rollings, Andrew and Morris, Dave., *Game Architecture and Design*. Coriolis Technology Press, Scottsdale, AZ, 2000.

[Ru03] Rucker, Rudy. *Software Engineering and Computer Games*. Addison Wesley, Essex, United Kingdom, 2003.