A FLEXIBLE AND EXPANDABLE ARCHITECTURE

FOR COMPUTER GAMES

by

Jeff Plummer

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

ARIZONA STATE UNIVERSITY

December 2004

ABSTRACT

Computer games have grown considerably in scale and complexity since their humble beginnings in the 1960s. Modern day computer games have reached incredible levels of realism, especially in areas like graphics, physical simulation, and artificial intelligence. However, despite significant advances in software engineering, the development of computer games generally does not employ state-of-the-art software engineering practices and tools.

This thesis describes an architecture for computer games as a System of Systems where the computer game itself is emergent. The proposed architecture follows a data centered framework where the independent components collaborate on a central data store. The architecture offers capabilities that are essential in overcoming challenges faced in building computer games that can enjoy modifiability, expandability, and maintainability traits. The architecture promotes component-based development (e.g., commercial off the shelf components) since the collaborating components have loose couplings, which in turn facilitates systematic design integration of System of Systems.

TABLE OF CONTENTS

Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

1 INTRODUCTION

1.1 Motivation

Electronic games are a billion dollar industry developing software system commonly reaching into the millions of lines of code ("3 Million"), but the development process remains very much unchanged from the early days of programming ("A $30 Billion Industry"). It's not unusual for development houses to move from the game idea directly to coding, where the success or failure depends almost entirely on the skill and experience level of the developers (Rollings 164-165). A base architecture that unifies the interaction between game subsystems and still allows for flexibility and expandability could greatly impact development the electronic entertainment industry.

1.1.1 The current Approach and Its Shortcomings

The current approach is to design and develop a custom architecture for each game. A game development house may carry over portions of a design from one game to another, but this is usually the result of individual experience rather than a formal design approach. So while skilled developers are still able to achieve the desired results, it is rarely on time and on schedule (Fristrom).

One problem with such an ad hoc approach to creating a game architecture is that quality attributes like flexibility and expandability are rarely incorporated in the design. For example ID™ software ended up rewriting almost all the code when moving from the game Quake™ 3 to Doom™ 3 (Sloan). Both are first person shooters, with the same game play. In fact the only noticeable difference is improved graphics. Since the game is primarily a graphical improvement, then the obvious culprit is the existing architecture didn't lend itself to expandability. ID's™ experience is definitely not unique. Countless

companies waste time rewriting music code, GUI code, etc. simply because the existing code doesn't fit into the new game.

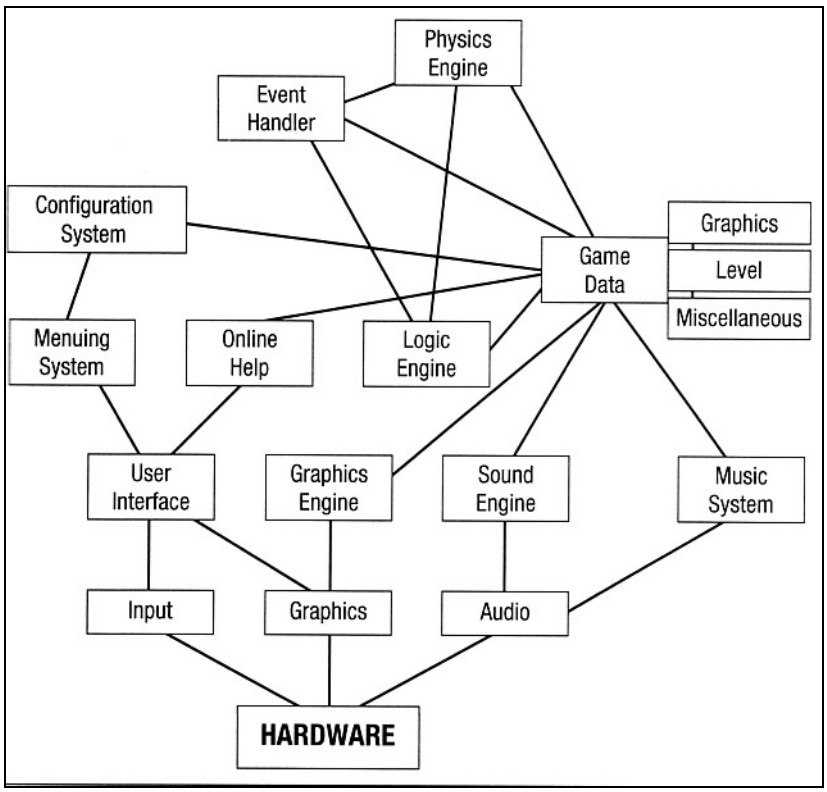

Figure 1 - Rollings' and Morris' Game Architecture

Rollings and Morris, the authors of *Game Architecture and Design*, reviewed existing game architectures, and attempted to map out a possible separation of logic (see Figure 1 above). While the component layout from Figure 1 may work for a game, I would argue the webbing of interrelated dependencies among subsystems would greatly limit the amount of expandability and re-use between game projects. A suitable architecture should not only have a logical separation of sub-systems, but also allow for

any of those sub-systems to be easily swapped out or modified without breaking the overall system.

Part of the reason most attempts at a game architecture have a great deal of interdependencies is because of underlying object-centric view of games (see Figure 2 below). Games have always been about game objects living in a virtual world. Game objects have their own behavior, draw themselves to the screen, and even make their own sounds. This view makes sense logically, and seems to follow the widely accepted object-oriented paradigm. This view, however, is starting to show its limitations as the complexity for such functionality as drawing and thinking continue to climb exponentially. Such complexity has made game objects unwieldy and difficult to design around.

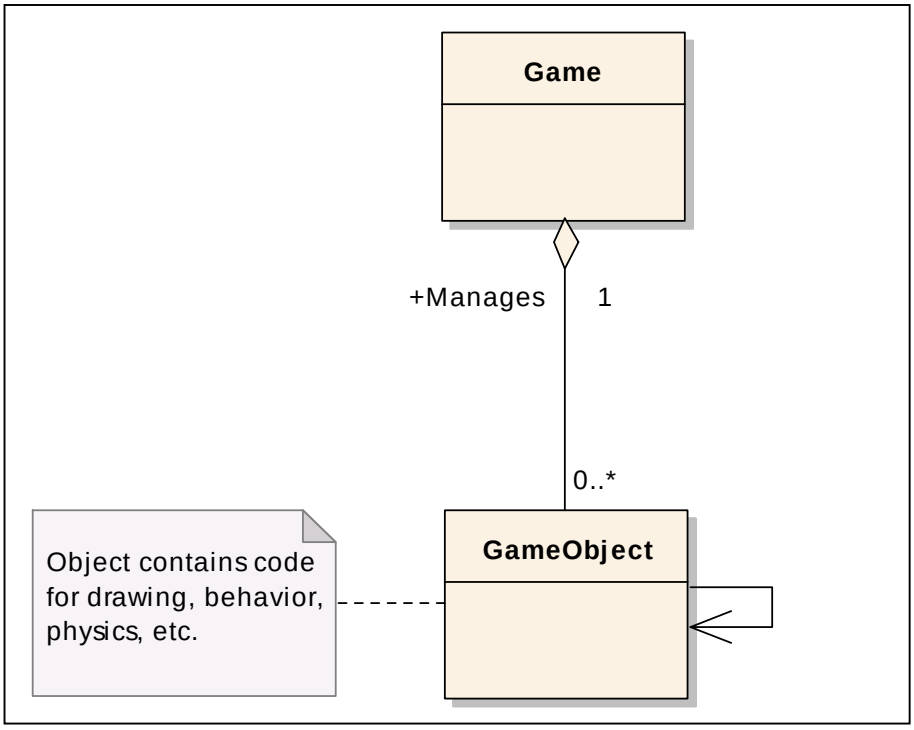

Figure 2 - Object Centric View of Games

1.1.2 The Migration to COTS

Software practices in games are undergoing a massive revolution. Games are approaching the production value of blockbuster movies, but without the same level of modularity and outsourcing. Movies are created by a several individual companies each specialized in areas like sound, special effects, etc. This level of separation of labor results in outstanding quality, and the ability to plan a timeline down to the actual camera shot. Games are just beginning this transition from 100% in-house code, to more of a Component Off the Shelf (COTS) based approach (Adolph).

Migration to COTS based systems is the first step in improving games on a massive scale. Allowing companies to focus on a single specialty means software technology can advance at a faster rate, and those advances are available for more games to use. While using COTS components can improve quality and time to develop (Alves et al. 1), staying with the current object-centric view means components are rarely more than functional libraries designed to help the object operate. Game objects are still responsible for all their own data manipulation including: graphics drawing, artificial intelligence (AI), sound, physics, etc. While games will still benefit from the technology COTS offers, it still means game objects are extremely large and complex. It also means game object developers must have a very strong knowledge about all the COTS components they are using to implement that object (See Figure 3 below).

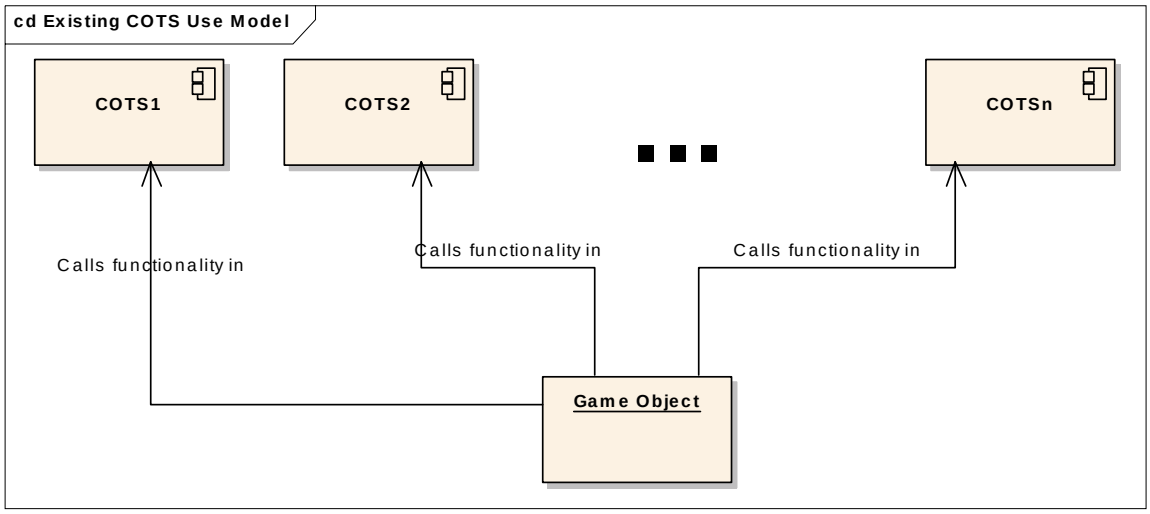

Figure 3 - Current Object Centered COTS Approach

The object-centric view also limits re-use, even when using COTS components. The object code is the least re-usable when moving between game projects, but it is the object code that contains the calls to the various components. So when moving from one project to another, developers will often have to re-write those same interactions. While a well-designed class hierarchy can mitigate much of that risk, the objects are still strongly tied to the COTS components they use. I propose there exists an architecture that can further increase code-reuse while reducing coupling.

1.1.3 Not a Game Engine

A common trend emerging in the game industry is the introduction to the all-encompassing COTS "game engine". Development houses can purchase very powerful "game engines", allowing the developers to develop a game using a commercially proven game framework. While this approach is an outstanding example of code re-use, it can

limit the flexibility of the developers to design the game of their choosing. "Game engines" can limit the developers in a variety of ways.

- Limits due to engine design – "Game engines" were built with an initial game in mind, and the completed design reflects this intent. Trying to use the UnrealEngine™, the game engine used to create the first person shooter game Unreal™, to create a console style football game may prove to be a very laborious task.
- Limits due to cost – "Game engines" can be very expensive. Top-tier game engines can cost in the hundreds of thousands of dollars ("3D Engines"). The decision to use such an engine means the game developed must be mass marketable in order to recoup that large initial investment. Unfortunately, in order to have mass market appeal the developer has significantly reduced options in what kind of game to create.

The intent of this thesis is to design at a higher level of abstraction than the design of game engine. This is not to say a reusable commercial game engine could not be developed using the proposed architecture, but the distinction should be made between an architecture and a fully fleshed out system design.

1.2 High Level Objectives and Goals

The main objective of this thesis is to design and prove there exists a software architecture that is both expandable enough to grow with the technology and flexible

enough to support the diverse world of games. Such an architecture would provide a starting place for game developers to begin from, and perhaps the start of a standardized communication between components used in a game system. A successful architecture will scale with the complexities of today's games, without sacrificing the developer's creative control over the game project. To achieve this rather lofty goal, the resulting architecture must fulfill the following requirements:

1.2.1 Architectural Requirement: Support COTS-Based Development

First the architecture must have strong separation of logic. The idea is to more completely separate the logic such that game subsystems can be independently developed and tested. This requirement is consistent with COTS based systems, and this thesis intends to continue with the COTS based approach.

In order to verify the resulting architecture meets this requirement it must be demonstratable that components can be independantly developed and tested. These components should be easy to integrate into a game application without a great deal of re-write on the part of the game. Ideally components will integrate in a similar yet logical fashion.

1.2.2 Architectural Requirement: Better Knowledge Localization

The architectural requirement of better knowledge localization exists because of the diverse capabilities required in games. Modern day games require outstanding graphics, realistic physics, mind-bending artificial intelligence, and theater quality audio. Even if the game developer is using COTS components to provide those capabilities, he/she must still acquire a large amount of domain knowledge in order to use the components

properly. The simple fact is game developers are forced to become experts in various technical fields when they should be focusing on developing gameplay.

The required level of domain knowledge is only going to increase as game technology advances, and an attempt to resolve this issue must be made soon. This thesis will endeavor to not only identify the commonalities between component interfaces, but also provide a design that minimizes the required component API understanding in order to use a COTS component.

In order to verify the resulting architecture meets this requirement the architecture should demonstrate a reduced API into the component itself. The technology components should also integrate into a game without requiring the game programmer understand the domain in order to use it. This eliminates the possibility of writing technology components a functional libraries.

1.2.3 Architectural Requirement: Flexibility / Modifiability

Flexibility is key to the future of game development. Due to rising production costs, the ability to mix and match re-usable software modules is critical to keeping costs down. The proposed architecture should be game genre and technology independent allowing developers to create a variety of games using various technologies. In order for this architecture to make an impact on the games industry, it must be flexible enough that any game project can use it.

In order to verify the resulting architecture meets this requirement it must be possible to demonstrate that very different games can be written using the final architecture.

1.2.4 Architectural Requirement: Expandability / Maintainability

Another critical architectural requirement, due to rising production costs, is expandability and maintainability. Successful games often have new incarnations with expanded game play and updated technology. For example, Blizzard's™ successful game Warcraft™ is currently on its third iteration with Warcraft™ 3. The new game features added game play elements like powerful heroes and beautiful 3D graphics, but the underlying game is still very similar. A successful architecture should easily allow for this type of game evolution.

In order to verify the resulting architecture meets this requirement it must be possible to demonstrate the architecture can easily support new or updated technology as well as new functionality. For example it should be easy for developers to move a 2D game to 3D graphics without a massive overhaul.

1.2.5 Performance and Other Quality Attributes are NOT Requirements

It may seem odd to not include performance as a key requirement when designing an architecture for a domain that demands such a high degree of performance. The reason for this stems from the belief that performance is far less significant at the inter-component communication level than it is within the subsystem itself. For example, the graphical rendering loop to draw the 10 million triangles of an object is far more significant to performance than the single inter-component communication telling the graphics system to draw the object. Performance will not be ignored in the design process, but the previously stated required quality attributes will carry a higher priority.

Other quality attributes, like reliability or portability, are also not ignored. The scope of this thesis, however, must be limited to qualities that can be verified and validated within the allotted time frame. Follow-up work would be to use the SEI's architectural tradeoff analysis method to determine how these other quality attributes are supported by this architecture. So for the purposes of this thesis, only those qualities deemed most important became a requirement.

1.3 Contributions

The primary contributions of this thesis are the following:

- A better understanding of games as systems. The artifacts created in this thesis will provide insight into what subsystems are involved in electronic games and their boundaries.
- An architecture that supports easy development and integration of COTS components for electronic games.
- An architecture that supports localization of domain knowledge, relieving the requirement for game developers to become experts in everything.
- An architecture that supports flexibility and expandability in game development by allowing developers to easily add/remove/modify game technology components.
- An architecture that support expandability and maintainability allowing developers to more easily expand a game into a future incarnation.

2 LITERATURE REVIEW

2.1 Current State of Game Development in Literature

There are currently dozens of books available on the subject of game development. Most, however, cover in great detail a specific topic in game development rather than an overall architecture. While these books definitely have their purposes, there doesn't exist any literature on how to properly organize these tidbits of knowledge.

Kevin Hawkin's and David Astle's book *OpenGL Game Programming* is a good example of a typical game programming book. The book covers some of the many graphics obstacles present in game development and how to use the OpenGL API. The book discusses 3-D math, lighting, texturing, transformations, and other topics of interest in programming 3-D graphics. After finishing this book the reader will have a solid understanding of graphics and the OpenGL API, but using this knowledge within the context of a complex system such as a game is still a mystery.

While the book is very well written, and covers the technical details involved in pushing pixels with OpenGL, it gives almost no architecture or design information. The book uses examples with a very monolithic design. A single game object will contain everything - graphics code, AI, physics, etc (See Figure 4 below). While this approach is fine for teaching the details of a game feature, it is HUGELY inadequate for a real game. The simple separation of logic at the class level just isn't enough for projects that can reach into the millions of lines of code.

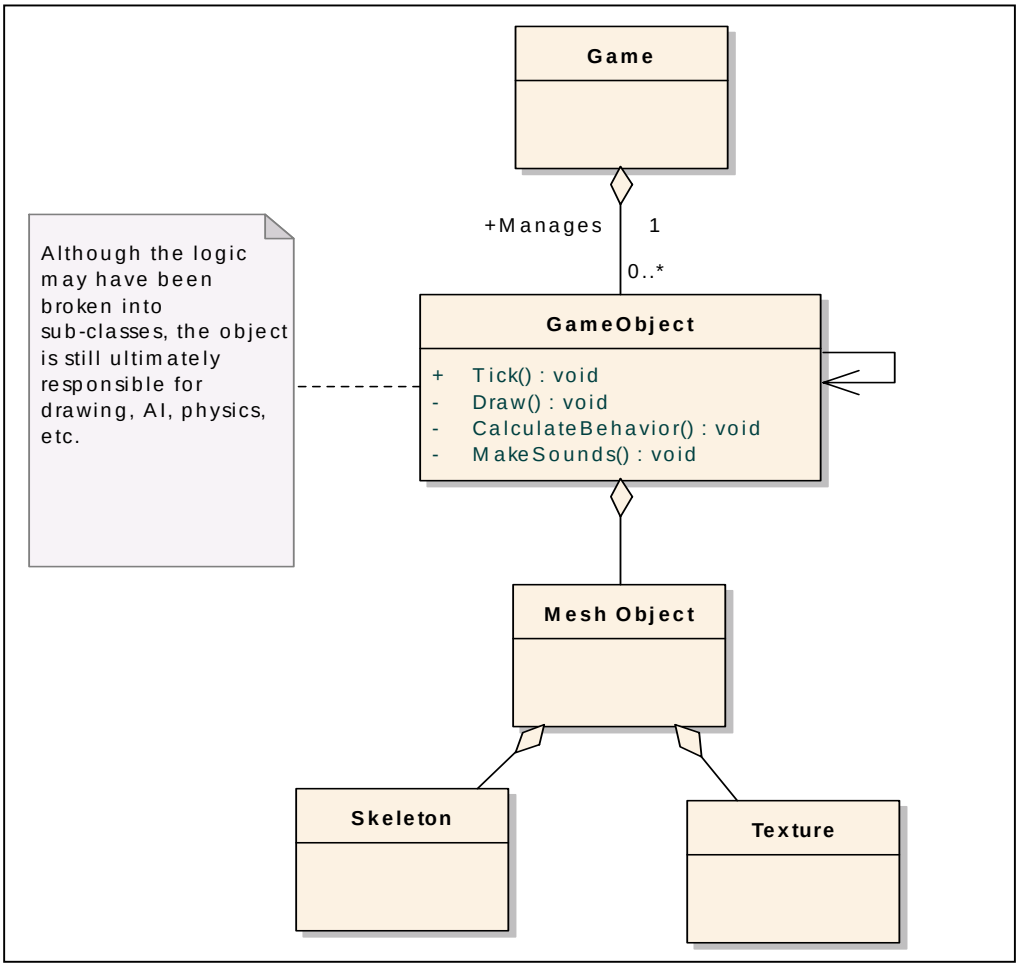

Figure 4 - Object/Class Level Separation of Logic

In order to see the many problems with such a microscopic approach to architecture, consider some of the issues game developers regularly face. First the design gives no insight into issues like portability, a very real concern for businesses interested in the various consoles as well as the PC. Next the code is not re-usable because objects are tightly coupled to their behavior. The design is neither flexible nor maintainable because this design is VERY tightly coupled and changes you make have the potential to affect the entire system.

Rudy Rucker's book *Software Engineering and Computer Games* makes an attempt to teach game development with a reusable architecture. The book creates a "Document/View" game framework. The emphasis is on the framework, as it is possible to create many different games by simply expanding the author's "pop" framework.

The book introduces how design patterns can be used in a game context, and why re-use should be important to a game developer. The author uses the document/view architecture to separate the data from the drawing code, thus allowing changes to the data without touching the visualization code.

While this framework has a great deal of flexibility in terms of game objects, it is still quite limited. AI and physics are still left inside the objects making changes to those areas very difficult. And while the graphics are somewhat separate from the objects, the author still uses direct access between the graphics and the data making the components both very dependant upon each other, and not quite staying true to the architectural model.

2.2 The Latest Book Trends in Game Development

The latest trend in game books is the "gems" like books. Books like *Game Programming Gems* and *AI Game Programming Wisdom* offer developer ready nuggets of wisdom. Snippets of code that offer very good solutions to difficult problems commonly found in game development. These books present low level solutions, usually in the form of a C++ class or two, that solve problems game programmers face everyday.

These books are an incredible resource because almost all their "gems" are architecture independent. They are solutions aimed squarely at helping the programmer,

not the system architect. So while the books are an excellent resource to any game developer, the solutions could not be strung together to form a coherent architecture. Developers can use the solution to solve a specific problem, but they may not understand WHERE the solution best fits into the overall system.

2.3 The First and Only Real Attempt at Game Architecture

Andrew Rollings's and Dave Morris's *Game Architecture and Design* is the only book on the market right now that discusses games in terms of their architecture. The book proposes to design around the quote by Dave Roderick, "A game is just a real-time database with a pretty front end." While that statement might seem correct, this thesis proposes the slightly modified statement – games are a system of systems operating on a database with a pretty front end.

The book gives an excellent introduction into the roots of game development and why architecture and software engineering practices have never really taken hold in this area of software. The authors attribute the lack of engineering practices to the origins and attitudes of game developers. Games originated from solo programmers who hand coded every line, and that solo attitude still prevails in the industry today. Not using third party components is still a point of pride for many developers.

While the authors provide an excellent history of the game development process, the book really doesn't spend much time on architecture (despite the fact that "architecture" is in the book's title). The book proposes an architecture for a game, but really doesn't provide any insight as to how the components communicate, or even why the proposed architecture is suitable and useful.

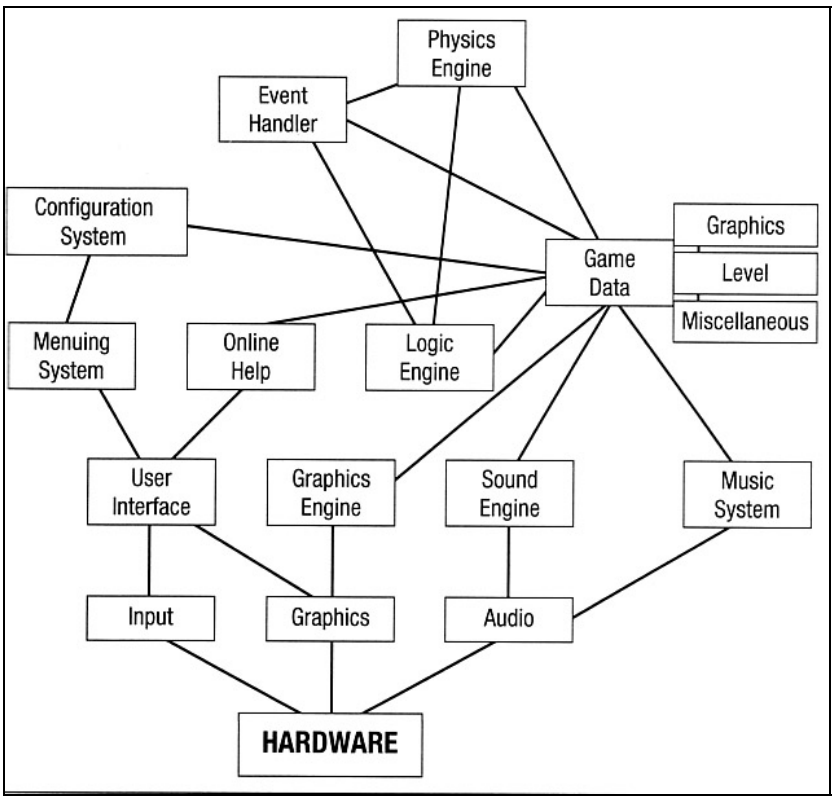

Figure 5 Rollings' and Morris' Game Architecture

2.4 Software Architecture

In order to properly design a flexible and expandable architecture for games, it is not only necessary to understand games, but also software architecture in general. Len Bass, Paul Clements, and Rick Kazman wrote *Software Architecture in Practice* as a very good introduction to software architecture. The book uses clear English to explain what an architecture is, as well as the concepts involved, including architectural style, reference models, and the reference architecture.

Software Architecture in Practice defines many of the quality attributes associated with an architecture, as well as what styles are best suited to each attribute. This book should prove useful when work begins on designing the proposed architecture. The reference offers a great deal of information that should help narrow down the search for architectural candidates.

Another book that provides some very useful insight in designing an architecture is *Designing Flexible Object-Oriented Systems with UML* by Charles Richter. This book provides many simple techniques to identify design flaws that can affect flexibility. The author teaches some guidelines to increase cohesion and decrease coupling, the advantages and disadvantages of class generalization and specialization, and an analysis of specialization versus aggregation. Richter also gives insight in how to analyze dynamic diagrams (e.g. sequence diagrams) for flexibility.

Richter's book should provide the litmus test for the flexibility in my design. He provides an informal, but effective way to quickly assess a design in terms of flexibility. Once the design has passed this informal inspection, a more formal approach can begin and the demo can be built.

3 THESIS METHODOLOGY

This thesis takes a pretty straightforward approach to arrive at the desired architecture. The first step is to analyze and understand games as software systems using standardize software engineering practices. Only looking at a few select games will scale this monumental task down significantly. The analysis will be further limited to identifying the functional modules and their interfaces. This level of analysis should provide enough of an understanding to begin the design work for the architecture.

The next step is to identify candidate architectural styles that have the quality attributes games require as identified in the "High Level Objectives and Goals" section of Chapter One. This step should yield architectural styles that should be considered when constructing potential architectures.

Once the preliminary research has been completed, the architecture design can begin. This involves incorporating various architectural styles into a design until the architecture can support not only the architectural requirements from Chapter One, but also the functionality Identified during the analysis phase. Through design trial and error, and architectural analysis techniques a proposed architecture should emerge.

After the proposed architecture has been designed, it is time to prove that it can work. The first step to proving the architecture will be to apply the architecture in the form of a simple design to the analyzed games. This will help to validate that the architecture can support the types of systems it was intended for. The next step is to actually build a game-like system to demonstrate the quality attributes. Unfortunately designing a commercial quality game to fully demonstrate the capabilities of the architecture are beyond the scope of this thesis. A demonstrative subset of game

functionality, however, will be put together into a prototype to show some of the more important features. A prototype will also have the added benefit of helping to refine the architecture into a more correct state, as well as identify some of it's limitations.

3.1 Analysis of Games as Software Systems

In order to design an architecture for the games of tomorrow, we must first understand the problems faced today. As noted in the literature review section, there exists very little documentation on the subject of architecture in games. Since more information is required, more creative approaches to analysis will be taken.

Since actual documentation on architecture in existing games is virtually non-existent, we will do the next best thing – understand the design of a game similar to existing games. The approach is simple. Treat an existing game as the customer requirements, and attempt to design a game that meets those requirements following standard software engineering practices. Performing this process for several games should provide a satisfactory understanding of what is required in an architecture to meet the needs of those existing games.

3.1.1 Selecting Games to Analyze

Since the goal of this thesis is to construct an architecture that will meet the needs of most electronic games, more than one game must be analyzed. In truth, such an architecture would require a thorough understanding of every possible game created. Due to the constraints of a temporal existence, this thesis will attempt to refine the search space into something more manageable.

Rather than analyze every existing game, existing games will be divided into categories where a single title could be selected to represent all games in that category. Fortunately the electronic games industry has already categorized titles into genres and we merely have to locate games representative of their genres. This approach should provide the best possible results given the time restrictions.

Game genres can be further divided into sub-categories like single-player vs. networked, 2-D vs. 3-D, etc. but I propose to show that these subdivisions are expansions of the same architecture. For example the differences between a 2-D game, and a 3-D game of the same genre should be localized in the components. However, the types of components and their interactions should remain the same. In the end I hope to show that a single architecture is capable of supporting all these genres.

3.1.1.1 Existing Game Genres

- **Fighting**

  The market was successfully introduced to fighting games in 1991 by Capcom and Street Fighter II. The opportunity to have fantastic heroes battle in hand to hand combat gave adolescent gamers the opportunity to connect to unique alter egos, and began the "golden age" of the arcade ("History of Arcade Games"). Fighting games are among the most simplistic in nature. They are meant to be simple, fast, and fun.

- **First Person Shooters (FPS)**

  First person shooters were invented in the 1992 by John Carmack and ID software with Wolfenstein 3D™ ("A Brief History"). The game

ushered in a new era of 3-D immersive worlds where players could explore, and experience the electronic universe in the first person.

This genre is probably the most diverse with games ranging from single player only, shoot to kill everything games like Doom™ and Quake™, to massively multiplayer universes like Halo™. First person shooters are almost always state of the art in terms of technology, and best noted for their outstanding graphics. Releasing a FPS using last years technology is a recipe for disaster in the retail market.

- **Platform**

  Platform games are the definitive arcade games. Icons like Super Mario Bros. ™ and Donkey Kong ™ were among the first to dominate the scene. Platform games require the player to navigate a character through various puzzles using a player's wit and skill with the joystick. Platform games are a relatively small market on the computer, but they still dominate the consoles with memorable characters like Lara Croft ™.

- **Strategy**

  The electronic strategy games of today are simply extensions of their board game ancestors. Strategy games typically involve intricate rule systems where player must master tactics and strategies rather than fast reflexes. Games range from the 2-D turn based classic Civilization™ to the 3-D real time masterpiece Warcraft™ III.

- **Role Playing**

Role playing is another genre that has its roots outside the electronic forum. Role playing games are a form of interactive fiction, where the player gets to play the role of one or more characters in the story. One of the staples of role playing games is character advancement. The character(s) the player controls will continue to grow in skills and abilities allowing the player to evolve a truly unique alter ego.

- **Sports**

Simply put, sports games are just the electronic versions of the real thing. Electronic sports games allow gamers to play the game without actually having train there whole lives to become professional athletes. Unhappy with the outcome of the super bowl, challenge your neighbor to a rematch in Madden 2002™.

Obviously there are games that do not fit into any of these genres or would be better described as a combination of genres, but these six categories arguably represent the bulk of electronic games available today.

3.1.1.2 Further Refinement – Isolate Important Properties

Unfortunately, properly analyzing even 6 games is too large of a task for the scope of this thesis. To ensure this further scaling has a minimal impact on the quality of the resulting architecture, I've decided to isolate the most important features. A minimum selection of games that covers those features will be chosen.

- **2D vs. 3D**

2D games are two dimensional games where the character exists in a two dimensional world. Platform games like Super Mario Brothers™ and strategy games like Starcraft™ are examples of 2D games.

3D games are games that take place in the third dimension. Here the distinction must be made between two dimensional games using 3D graphics, like Warcraft ™ 3 and games with fully three dimensional worlds like Quake™. For this thesis it is important to select a game in the later category, because it is important to maximize the differences in the game components. Fully three dimensional worlds require different physics, AI, as well as 3D graphics.

All games fall under one of these two categories, so the final selection must include one game from each category.

- **Non-Networked vs. Networked vs. Massively Multiplayer**

Non-networked games are games that exist on only one machine. Code and data does not need to be distributed across a network while the game is playing. Almost all games offer this style of play, allowing the human player to compete against computer opponents on a single machine.

Networked games are games where human players can compete against other human players over a network. Most games of this sort use

the simple client/server model and usually have a set maximum number of players (clients) per game.

Massively Multiplayer Online Games (MMOG) have been around for a while in many text based multi-user dungeons or MUDs, but have become very popular in the mainstream with the 3D dungeon romp - Everquest™. MMOGs allow thousands of players to exist persistently in a virtual world. Unfortunately due to the scope of this thesis, MMOGs will not be covered, but definitely represent an area that should be covered in future research.

- **AI – Single Entity vs. Managed or Team**

  Artificial intelligence in games can be broken into two very simple categories. Games with single entity intelligence are games where each game object has its own AI and behaves relative to its own situation. There is no mastermind or general coordinating the actions of the objects to form an overall strategy.

  Managed or Team AI games expand on the single entity AI model and add the concept of collaboration between objects. Objects still have their own intelligence, but a new layer has been added that can view the game in terms of tactics and strategy.

3.1.2 The Selected Games for Analysis

After a great deal of review, the search has been narrowed down to two games that exist in to different genres and cover all the important properties. While these two

games cannot fully represent all possible electronic games, these two games should provide a solid foundation given the time constraints and scope of this thesis. This foundation should be adequate to isolate many of the component interactions and support the design of an architecture that could support the needs of most games.

Starcraft™ by Blizzard Entertainment

The first game chosen for analysis is the award winning Starcraft™ by Blizzard Entertainment™. The game features a 2D isometric view and some of the best game play ever. The game was released in 1998, and has become the yard stick all other real-time strategy (RTS) games are measured by. For analysis purposes the game was chosen because it is two dimensional, offers solid non-networked or single player game play, and a managed AI system.

Figure 6 - Screenshot from the Game Starcraft

Unreal Tournament

Unreal Tournament(tm) by Epic Games Inc. is easily one of the best networked first person shooters ever. The game offers up to 16 players a chance decimate each other in a futuristic combat arena. Players enter UT's 3D proving grounds and become the combatant, taking control of a single character. While newer iterations of UT have been developed since UT was released in 1999 (“Unreal” 1), see screenshots below, they are primarily upgrades in technology. Unreal Tournament(tm) was chosen because it features 3D graphics, networked play, and any AI is primarily centered around a single entity.

Figure 7 - Screenshot from Unreal Tournament

Figure 8 - Screenshot Unreal Tournament 2004

3.1.3 Analyzing the Games

Having selected a seemingly diverse pair of representative games, we can begin the analysis process. By designing an architecture capable of supporting these two dissimilar games, it is the hope of this thesis that the architecture can support the development of just about any type of game. The analysis will pretty much follow the standard software engineering practices for system development.

The process begins with understanding the systems requirements, which can be done by treating the final game as the customer requirements. From the final game, use cases can be derived and reviewed for further analysis. From that point, we can begin to find the subsystem interactions that need to exist in the proposed design.

3.1.3.1 Analyzing Starcraft™ Requirements with Use-Cases

The first part of analysis is to understand the requirements of the system we are trying to build, or in our case merely understand. Since our requirements are based on a finished piece of software, requirements and use-cases can be harvested from the game's manual and from playing the game itself. After a first pass of studying the manual and actually playing the game, I came up with the following use case diagram:

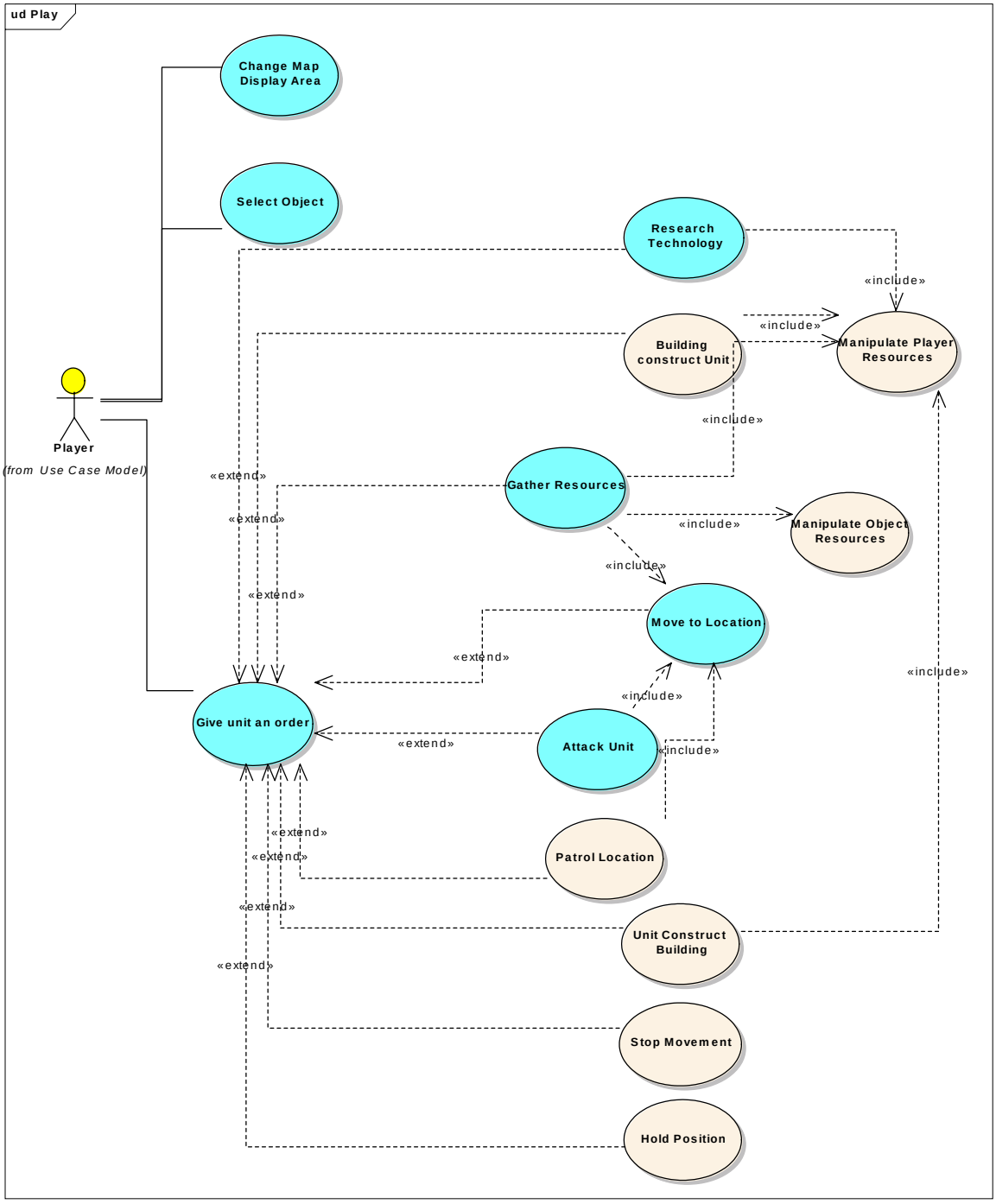

Figure 9 - Playing Starcraft Use Case Diagram

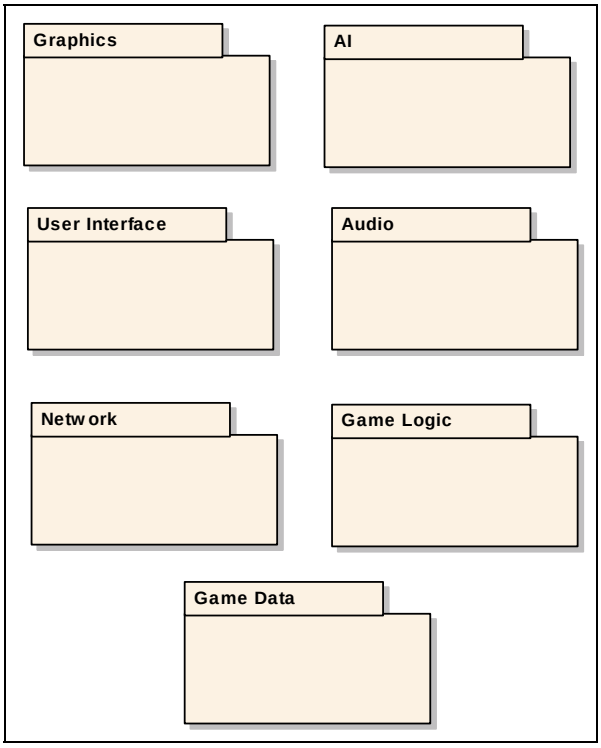

Figure 10 - Logical Modules

Based on the details of the use cases in the diagram in Figure 9 above, we can begin to identify the functional modules involved in the game of Starcraft™. 2D graphics functionality is needed to render the game objects, the user interface functional module will capture the players input, and so on. While Figure 10 above is not meant to show the physical separation of subsystems as a component diagram would, it does show at a high level what kinds of functionality are needed within the game Starcraft™.

Before we begin analyzing the types of sub-system interactions that need to exist in the system, however, we must first isolate which use-cases will be used to guide the analysis. The final analysis of this game, located in appendix A, has very many use-cases. Due to the time and scope constraints on this thesis, it would be impossible to

fully explore them all. To ensure the research is still adequate I based the selections on some very simple criteria.

First the use-case must be fundamentally important to the game. Since our original game title selection was based on each game being representative of many other games, there is no point wasting time analyzing actions that don't represent those other games. Second, the use-case must require multiple sub-systems to collaborate. Since the goal of this next phase is to understand logical module interactions, we can eliminate the trivial use-cases. In all diagrams, use-cases selected for further elaboration have been colored a light blue.

3.1.3.2 Understanding the Sub-System Interaction

Having selected multiple use-cases for further analysis we can begin trying to understand the communication between logical modules required to realize those use cases. Consider the *Select Object* use-case from Figure 12. "User left-clicks the mouse button while the cursor is placed directly over a selectable object in the main view. The selected object is marked with a green circle and is ready to receive orders." In applications using the "document/view" architecture, this use-case is almost trivial to implement. The view receives the mouse click, determines which object was clicked based on its screen coordinates, and sends the click event to the object for processing. The object can then decide to draw a circle around itself or whatever.

At this stage we have not yet decided anything further about the architecture, so the focus is not to design the interactions as in the document/view example. Instead the goal at the analysis phase is to understand the "kinds" of interactions, not how those

interactions will actually be designed. Consider that same *Select Object* use-case following a model-view-controller approach. The "kinds" of component interactions that need to take place are still the same.

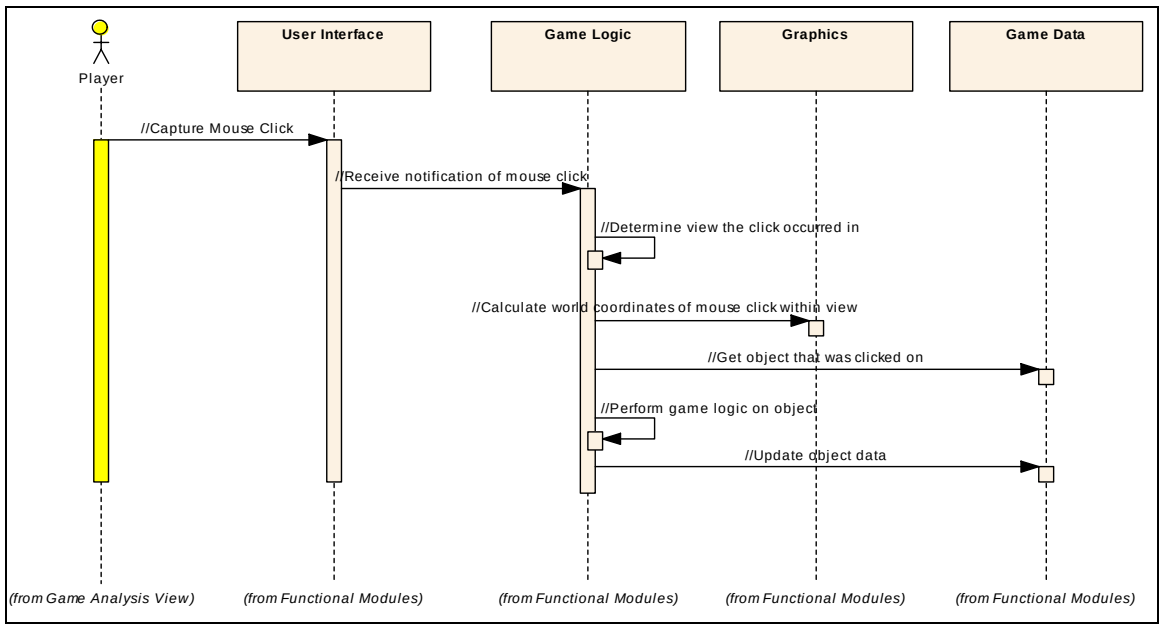

Figure 11 - Select Object (Subsystem interactions)

Figure 11 above shows an example sub-system interaction. The trick is to understand that the diagram above is NOT the design. The "Select Object" sequence diagram above shows that the graphics system is involved in routing mouse clicks to the proper object. It does not necessarily mean the Game Logic system calls the graphics system. Perhaps current screen position is part of the object data set by the graphics system at a different time. The important thing to note is that in order to determine which object was clicked, the UI and graphics systems are involved. Once all the important use cases have been further analyzed to isolate the kinds of interactions we can begin creating a potential design. To see other component sequences refer to appendix A.

3.2 Identify Candidate Architectural Styles

The next step before we can design an architecture is to consider the architectural styles that have already been shown to exhibit the quality attributes games require. In its simplest form an architectural style is a set of components, their constraints, and the constraints on their communication (Bass et al. 25). By incorporating well-understood architectural patterns, we are more likely to achieve a hybrid design that will achieve our goals.

Several architectural styles appear to have some of the desired quality attributes, and will be reviewed.

3.2.1 Layered

The layered architectural style divides system functionality hierarchical layers where each layer provides services to the layers above and below it. The layered approach tends to promote re-use by keeping the application specific code at the top most levels, allowing developers to re-use the framework below (Duffy). Re-use is directly tied to the flexibility and modifiability requirements of this thesis.

3.2.2 Data-Centered

The data-centered style is essentially a centralized data store with independent clients connecting to operate on the data. Data-centered styles offer an easy way integrate different systems because the clients are independent of each other, and the data store is independent of the clients (Bass et al. 95-96)

3.2.3 Independent Components

Independent processes communicating via messages define the independent component architecture. Components register the kinds of information they can process, and communicate through messages (Bass et al. 101-102). One interesting advantage of the independent component styles is that all components need not exist. The decoupled communication system is such that published messages may not have any subscribers. A well-designed system could add and remove functionality at will.

3.2.4 Data Flow

Data flow architectural styles like pipe and filter tend to offer a great deal of re-use, and are generally easy to maintain and expand (Calvert). By focusing on incremental transformations of data, systems are very simple to understand and change. Systems can be easily expanded or modified simply by plugging in new or different data processing components (Bass et al. 96-97). The notion of effortlessly expanding games by extending the chain of data processors is very appealing.

3.2.5 System of Systems

The system of systems architecture is the part of engineering work being done to integrate multiple complex systems. The SoS approach is interesting because both games and enterprise applications must integrate systems of entirely different domains. Graphics, physics, AI, etc. are entirely unique domains being used together in a single application. Another interesting aspect of SoS is the point of view that a system is

emergent from the integration of the individual subsystems (“Definitions”). In other words a game is the result of integrating an AI system, a physics system, etc.

Such a unique view matches one of our initial requirements of domain knowledge localization. So if a graphics engine is a complete system, and the game is actually the result of the graphics engine working with other systems, it may be possible to keep the graphics details hidden from the game itself.

3.3 Architecture Design

At this point both games selected for study have been analyzed such that we have descent understanding of what logical modules exist, and the kinds of interactions that must occur to perform the game functionality. The next step is to actually determine the overall system layout, and how the different subsystem interaction will occur. By incorporating the various architectural styles noted in Section 3.2 of this thesis, we will design an architecture that should meet the requirements we have laid out, as well as support the functional needs common to games.

3.3.1 Choosing a Topology

The first step to developing an architecture is deciding upon a topology. The topology is the over-all layout of the system, and has significant impact in terms of modifiability and reusability. The topology determines what logical systems are connected; thereby setting what coupling may exist. The plan is to see how different architectural styles can be applied to the logical modules in games, and determine the affects it might have on the quality attributes the desired architecture requires. While only one topology will be selected (or perhaps a hybrid of a select few) for further study,

those that weren't selected may provide some ideas that can still be incorporated into the final architecture.

3.3.1.1 Layered Architectural Style

The first major topology considered was the layered architectural style. The layered architectural style tends to offer many quality attributes, of which re-usability and modifiability are most import to our goals. Looking at the simple diagram in Figure 12 below, the game specific code is localized in the top-most layer. By localizing the game specific logic to a single layer, new games can be created re-using the layers below.

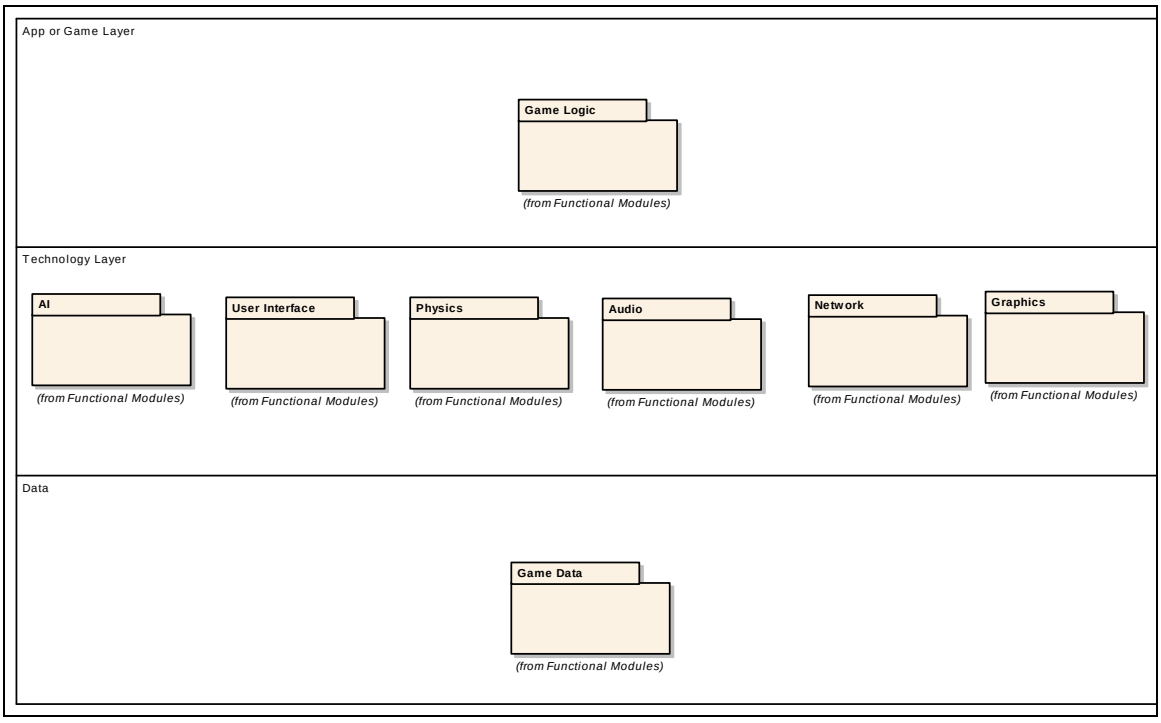

Figure 12- A Simple Layered Architecture

The above "start" of an architecture has many problems that ultimately led to the dismissal of this topology. First, while the architectural approach offers some re-usability

between games, it does not appear to offer much re-usability between different types of games (i.e. different technologies). This view of a layered approach doesn't offer much insight in how a developer might move from a 2D platform game, to a 3D first person shooter. Such a change would require a very different graphics module, a very different AI module, as well as requiring additional modules that probably wouldn't exist in a 2D platform game (like physics).

Obviously this approach could be refined with layers further divided, but the underlying problem still exists. It isn't just the game code that is likely to change, but the technology modules as well. Also due to the fact that different sets of logical modules may be needed for different games, with potentially different module interactions, perhaps layering does not isolate likely changes in the best possible way.

3.3.1.2 Data Flow Architectural Style

Data flow architectures offer a great deal of flexibility in that data processors can be added at will. The problem becomes very apparent, however, when you look at the game modules in this layout (see Figure 13 below). Game components operate on very different data. The data pipe connecting these logical modules would have to be so broad that each module might spend significant overhead parsing and filtering out the large amount of data that isn't used (Calvert).

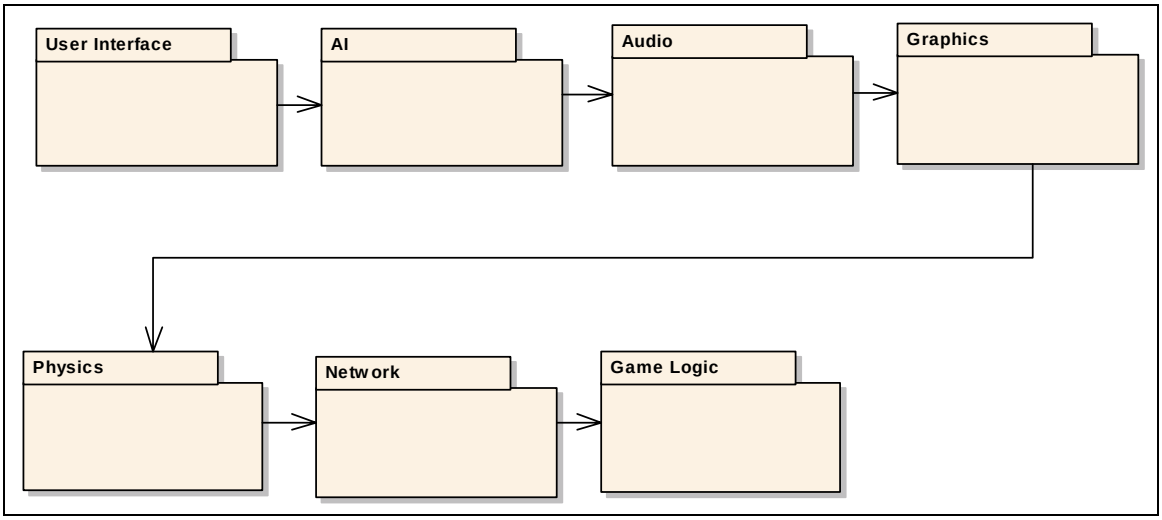

Figure 13- Data Flow

The data flow architecture does, however, present some interesting options for the architecture at the component level. Figure 14 below shows a simple example of how an AI component could be implemented using the data flow architectural style. Unfortunately this thesis is focusing on the architecture at the game system level, so this concept will be left for future research.

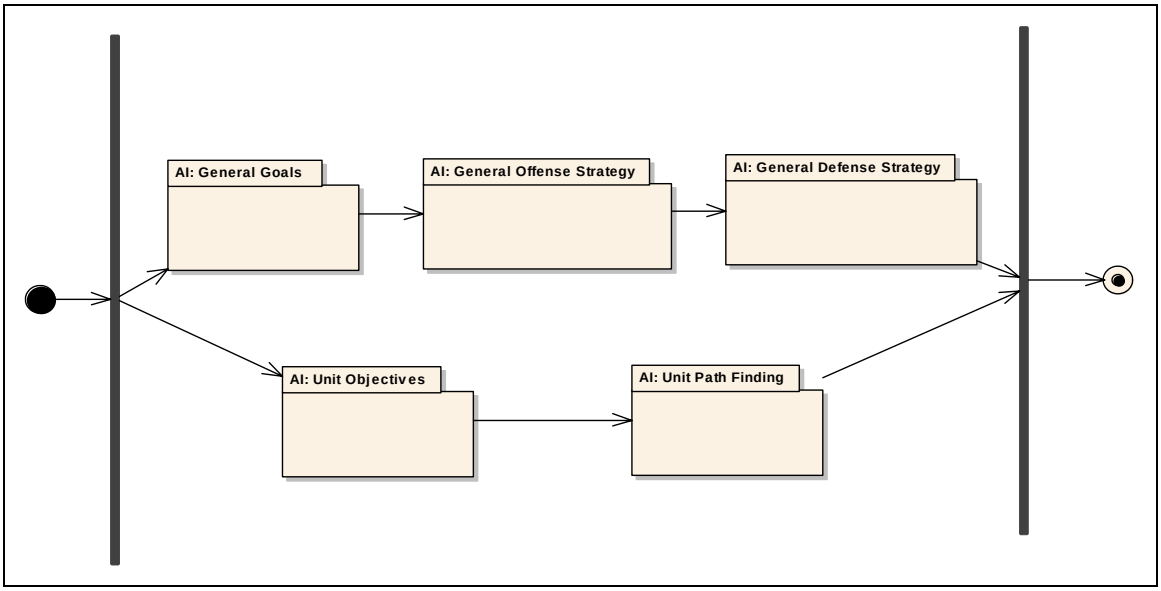

Figure 14- Data Flow at the Component Level (AI)

3.3.1.3 Data Centered Architectural Style

Another major topology of interest is the data centered architectural style. A data centered approach is typically used to create data integrability. Functional modules are less strongly coupled, but often at a cost in performance (Bass et al. 96). The data centered approach minimizes many of the risks identified in the layered approach. First, the logical modules do not have any direct interaction with each other mitigating the issue of changing technology. Changing from a 2D graphics logical module to a 3D graphics logical module should not break the workings of the other sub-systems.

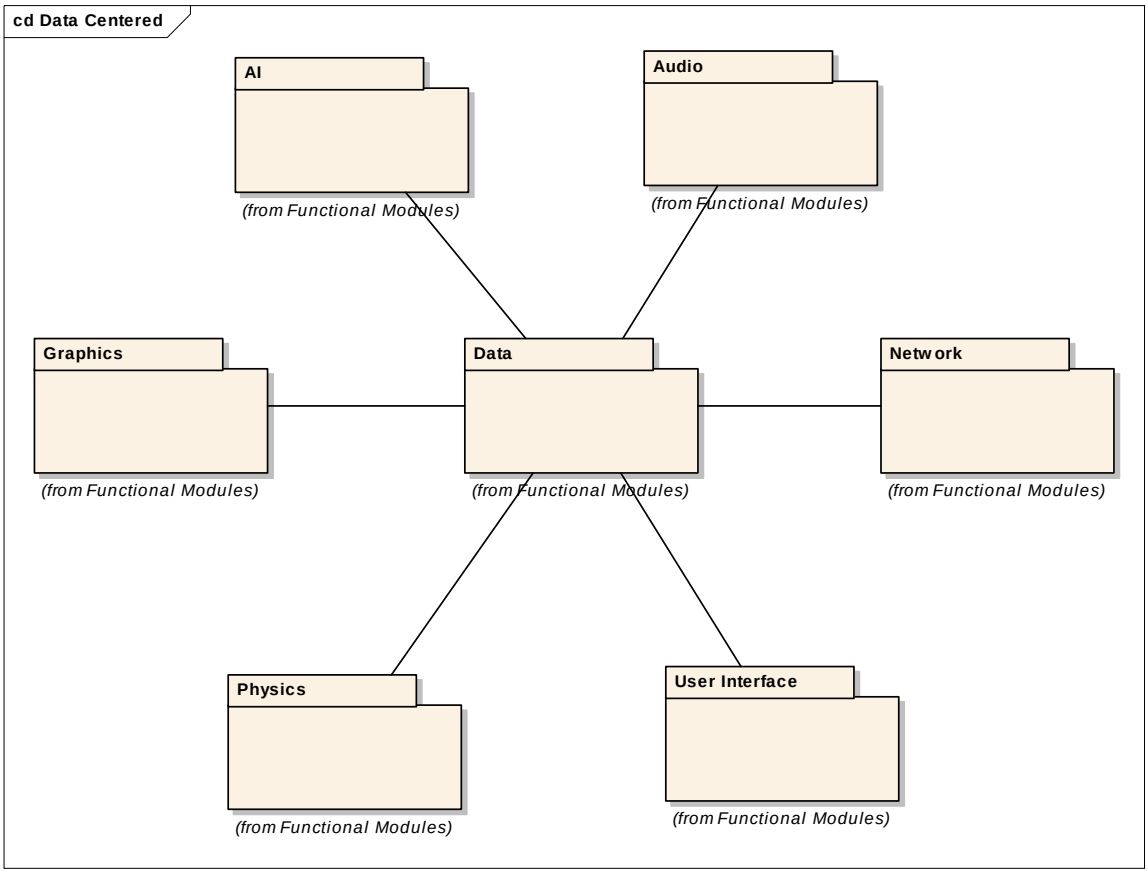

Figure 15 – Data Centered

There is still the issue of modifiability at the game level. Figure 16 above does not give any indication of how the developer can minimize the amount of change when moving from one game to another. In the layered approach, game specific code was localized to a single layer, making it easy for developers to move between similar game projects, while the data centered topology doesn't provide much insight as to how game specific code could be localized. This issue will continue to be worked as the architecture is further fleshed out.

While using a data centered approach does offer many architectural benefits, it drastically impacts how game functionality can be achieved. Consider the use case diagram shown earlier in Figure 11 - Select Object (Subsystem interactions). This simple act of clicking a button changes dramatically because there is no direct association between the User Interface and Graphics logical modules. Both figures demonstrate the same functionality, but the data centered topology has placed some constraints on the way it can be realized.

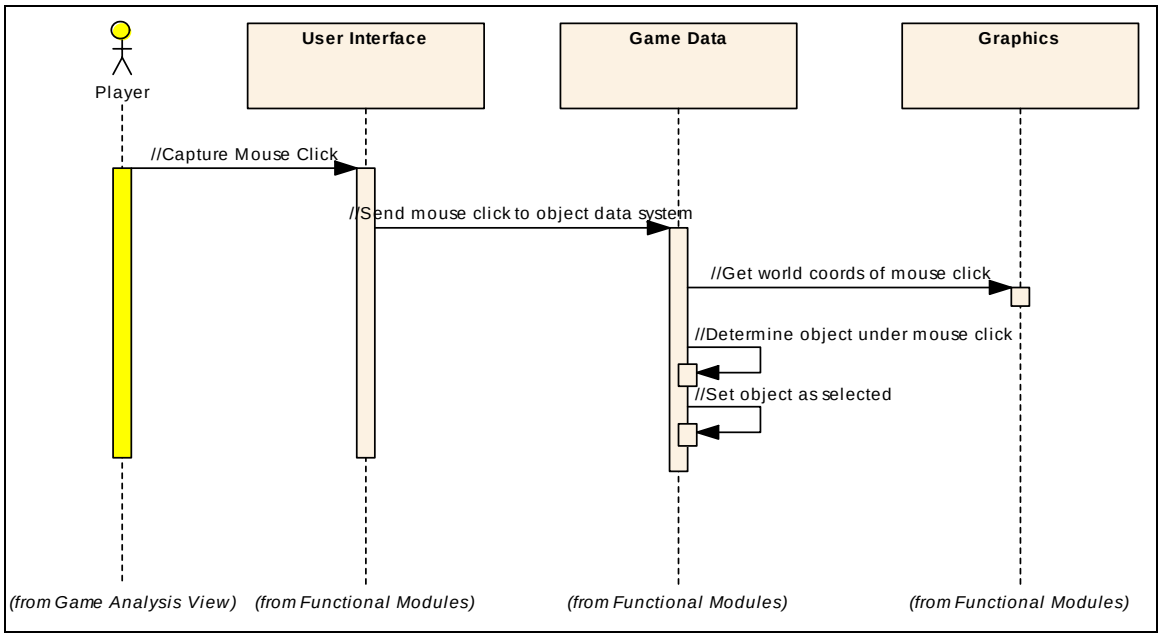

Figure 16 – Select Object (Logical Module Interactions – Data Centered)

3.3.1.4 Independent Components Architectural Style

A game using the independent component architectural style can have any arbitrary topology because of the style's restrictions on communication. Regardless of the layout, independent components remain decoupled because they communicate via messages

rather than making function calls. The interesting aspect of this approach is components don't know who they are sending to, and don't necessarily need to wait for a response. This not only means new components can be added/replaced, but partial systems can be built with components missing. This means systems can be put together even before all the components are built, greatly increasing the ability for individual components to be independently developed.

Figure 17 below shows an example of how a game could potentially be put together using the independent component architectural style. This rough sketch highlights some of the strengths and weaknesses of this approach for a game system. The user interface component as an independent component communicating via messages makes perfect sense because user interactions really are asynchronous events. When you start to move to how other components like graphics and AI interact with data, event based communication makes less sense. Every cycle some game data must be drawn, must perform AI, and must have some form of game logic applied to it. The overhead of routing and translating messages becomes significant when the number of messages approaches some threshold. Due to the sheer volume of data involved in games, and the synchronous nature between some of the subsystems and the data, perhaps independent components is not the best architectural style for this domain. It would, however, be an interesting research project to see just how much the messaging overhead would affect systems with synchronous interactions like games.

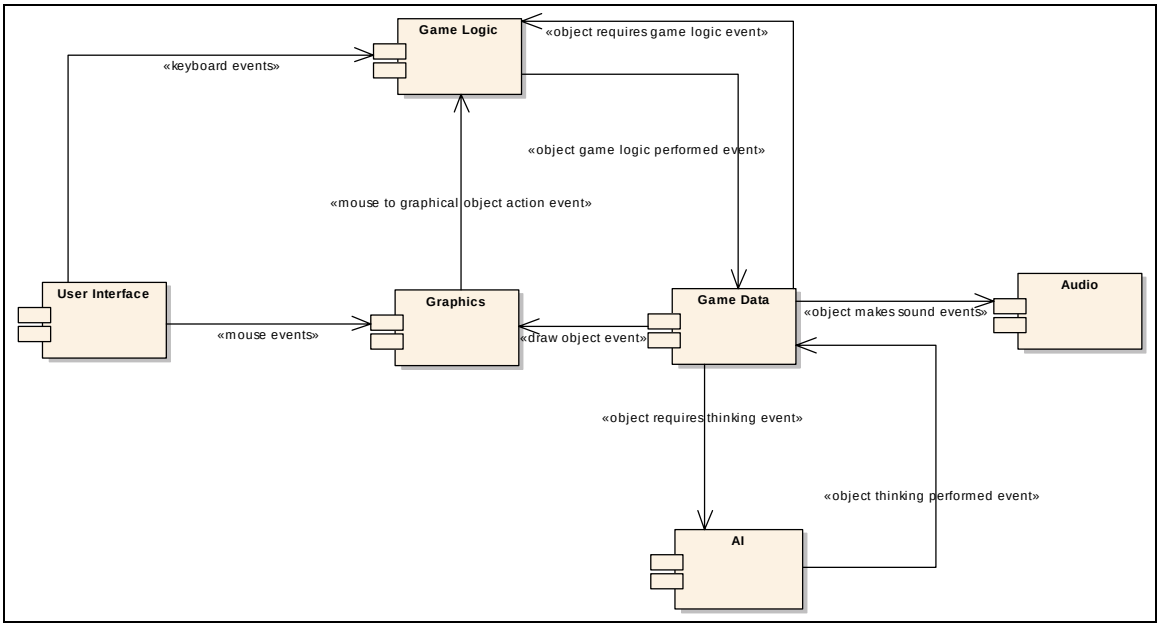

Figure 17- Independent Components

The qualities achieved by independent components should not be completely discarded simply because this particular style may not be the best choice. The ability to put together an incomplete system with components missing is a very useful idea. Consider a development scenario where the graphics system has not been selected, or is behind schedule. An incomplete game system consisting of the game logic, data, and AI could continue to be worked. So even though one of the subsystems cannot be used, the game as a whole can continue integration work.

3.3.1.5 System of Systems

The system of systems was rejected for the same reason the independent components architectural style was rejected. Event based communication is just too inefficient for some of the interactions. The time researching the system of systems perspective,

however, was definitely not wasted. The notion that the desired system, a game in our case, can be emergent as a result of the collaboration of other systems is a very interesting idea ("Definition"). Just because the arbitrary topology and method of communication are ineffective for this thesis, doesn't mean the underlying idea can't be used.

3.3.2 Making the Topology Choice

Choosing a topology forms the structure from which the architecture will evolve. It determines how systems can grow and change, and has significant impact on the qualities the final architecture will exhibit (Bass et al. 105-107). The research has shown that arbitrary topologies appear to place too much overhead on communication in order to keep the subsystems truly independent, a key requirement for this thesis. Even though performance was not one of the key requirements for this architecture, other approaches are still able to meet the requirements without imposing such a high performance cost.

The data flow architecture appears to not be the best choice because the logical domains are just too different. Trying to design a universal data pipe for all the data involved in games doesn't seem like the correct approach. The analysis performed seems to suggest that the data flow architectural style is just the not the best starting point for the system level of abstraction.

The layered approach is more structured and could possibly provide better performance than the other less structured topologies. The layered topology, however, cannot easily abstract functionality in a way that minimizes the effects of changes in technology. Part of reasoning behind this thesis is the belief that changing technology

has a greater impact on work required and the level of modifiability, than the ability to swap out the coded game logic. Game technology is moving at an astounding rate, and gamers often only buy games with the latest technology. This means developers must constantly upgrade the technology modules or suffer in sales. It would be possible to place each logical system in its own layer, but doing so essentially emulates the data flow architecture and all its problems.

Ultimately, the data centered topology was chosen for further analysis because it showed the greatest mix of flexibility and performance. The other approaches may have offered some truly desirable characteristics, but had significant inherent disadvantages that would be difficult to overcome. The data centered approach still allows for sub-system independence, but allows a direct communication to the game data. At this point it seems the data-centered layout offers the best chance at designing an architecture that meets all of the proposed requirements.

By moving forward with the data-centered topology this thesis is placing a higher priority on providing flexibility in technology usage than on re-using an existing framework. This prioritization is also matches one of the original goals for this thesis – supporting COTS-based development. Modern game complexity is just too large for single development house to create it all. An architecture design that supports easier integration of COTS technology will likely better serve the industry. It should also be noted that choosing to start from a data-centered topology does not necessarily restrict the use of a different topology at a different level of abstraction. For example, it may be possible to use both a layered and data centered topologies as in Figure 18 below.

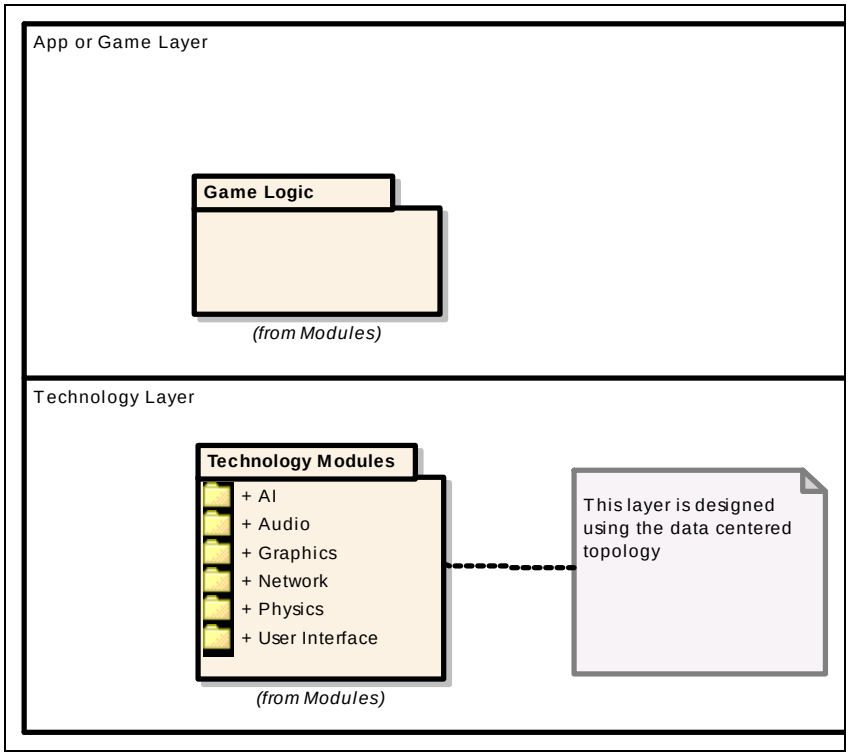

Figure 18 - Layered and Data-Centered

3.3.3 Choosing a Style of Communication

Once the overall topology of the logical modules has been created, the method of communication must be designed. We have decided what modules can communicate, but it has not been decided how that communication will work. Following with the data centered topology there are two common models available for the communication between clients and the data.

3.3.3.1 Repository

The first is the repository model where data resides in a passive repository. Because the data repository is passive, clients are responsible responsible for pulling the

data and determining if it has changed. The repository is probably the simplest to understand because the data store is essential a database answering queries.

The only real downside to this methodology is the increased traffic between a client and the data store. The client is requesting data for processing even if the data has not changed.

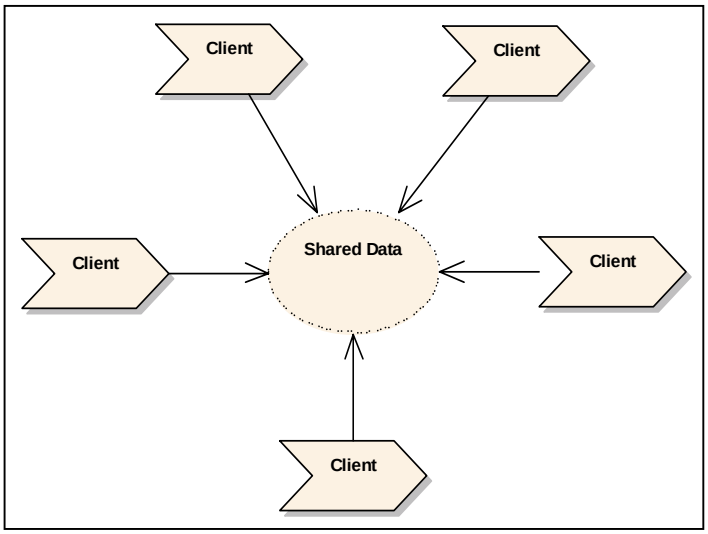

Figure 19 - Repository

3.3.3.2 Blackboard

The second common model for communication is the blackboard method. In this model the data repository is active and sends update messages to clients informing them of the updates to the data (Bass et al. 95). The blackboard methodology is an attempt at reducing the amount of communication and as a form to keep the clients synchronized.

3.3.3.3 Making the Communications Choice

For the games domain the repository model was chosen because it makes the most sense logically. First, the increased communication between the client and the data store is less of a concern because both will likely reside on the same computer. Second, a domain-specific module will need to operate on an object whether it's data has changed or not. For example, a graphics engine will need to draw a visible object even if it's position hasn't changed since the last time it was drawn. Lastly, the repository model also has the benefit of keeping all the data localized in the one area meaning domain-specific modules don't need to maintain local copies of the data.

3.3.4 Synchronicity

Synchronicity is how the data and control flow through the functional modules. Because synchronicity is tightly tied to the topology and method of communication we have already eliminated some possibilities. For example, since we have chosen not to use the blackboard method of communication asynchronous methods of synchronization may not be the best choice. Fortunately traditional approaches to game development already use a method of “ticking” game objects, proving that games can be built using a synchronous approach.

3.3.4.1 Synchrous at the Object Level

Synchronization at the object level is where all of the object's functionality is completed before moving on to the next object. In other words an object performs AI,

draws itself, etc. then moves on to the next object. If you stick with the paradigm that a game is just a bunch of game objects then this method makes sense.

3.3.4.2 Batch Synchronization

Batch synchronization is the case where a large group of objects are processesed completely before moving on to the next group. A game example might be that all objects perform their AI calculations before they are drawn. This approach starts to make sense the more complex the specific functionality becomes.

3.3.4.3 Hybrid Synchronization

Current approaches to game development today often use a mix of synchronous approaches at the object level and component level. “Ticking” an object may result in the object performing AI, and making sound, while drawing the objects may be done as an entire batch. This probably a result as games evolved. In early days, games were simple enough that synchronization at the object level. As technology has grown more complex, it's often easier to write an entire “engine” to perform things like graphical rendering as a batch operation (Rollings 453-454).

3.3.4.4 Making the Synchronicity Choice

The choice to move ahead with batch synchronization was made for several important reasons. First, synchronization at the object level using the data centered topology with a passive repository does not make a lot of sense. Synchronizing at the object level defeats the whole purpose of having functional modules operating independently around a common data store. Having each functional module operate on

the relevant objects and then moving to the next functional module does. Second, one of the main reasons for this research is to deal with the fact the domain-specific processing is becoming more and more complex. And as games are already beginning to see, it is easier to handle complex calculations when operating as an "engine" performing a specific type of functionality all at once.

3.4 The Idea – System of Systems Philosophy

Having performed a great deal of research in both games and software architecture I have come to really like the system of systems philosophy. While the common concept of SoS appears to have too many performance issues to make it viable for games, the underlying idea is sound. The notion that that independent and complete systems are collaborating and result in an emergent system is very powerful.

Designing a game as a collaboration of independent game subsystems has a great deal of potential. First, development and test are simplified because dependencies between sub-systems are eliminated. Second, incorporating the subsystems into a game can potentially become much, much simpler. Because a logical module is a complete system, the game is not using the module as a programming library with game specific function calls. Instead the logical module is configured to behave as a system that will result in what the desired game needs. So using a game subsystem becomes a matter of configuring a system, rather than learning and using a domain-specific programming API. The proposed architecture will attempt to incorporate this simple idea, and possibly create a new approach to developing games.

4 THE PROPOSED ARCHITECTURE (and a Simple Design)

The proposed architecture takes a step back from looking at games as a system of game objects, and looks at them more as a data centered System of Systems (SoS). An architecture where external systems (graphics, AI, etc.) work together toward a common goal, and the game is formed as the collaboration between those systems working on the same data set. This chapter will present the architecture and a simple design using the proposed architecture.

4.1 The Data-Centered System of Systems Topology

The architectural structure is represent below in Figure 19. Domain-specific systems operate independently on a shared collection of data. The domain-specific systems are responsible for requesting data to operate on, and update. Another issue to note, but will be further explained, is the domain-specific systems can store domain-specific data related to game objects within the common data store.

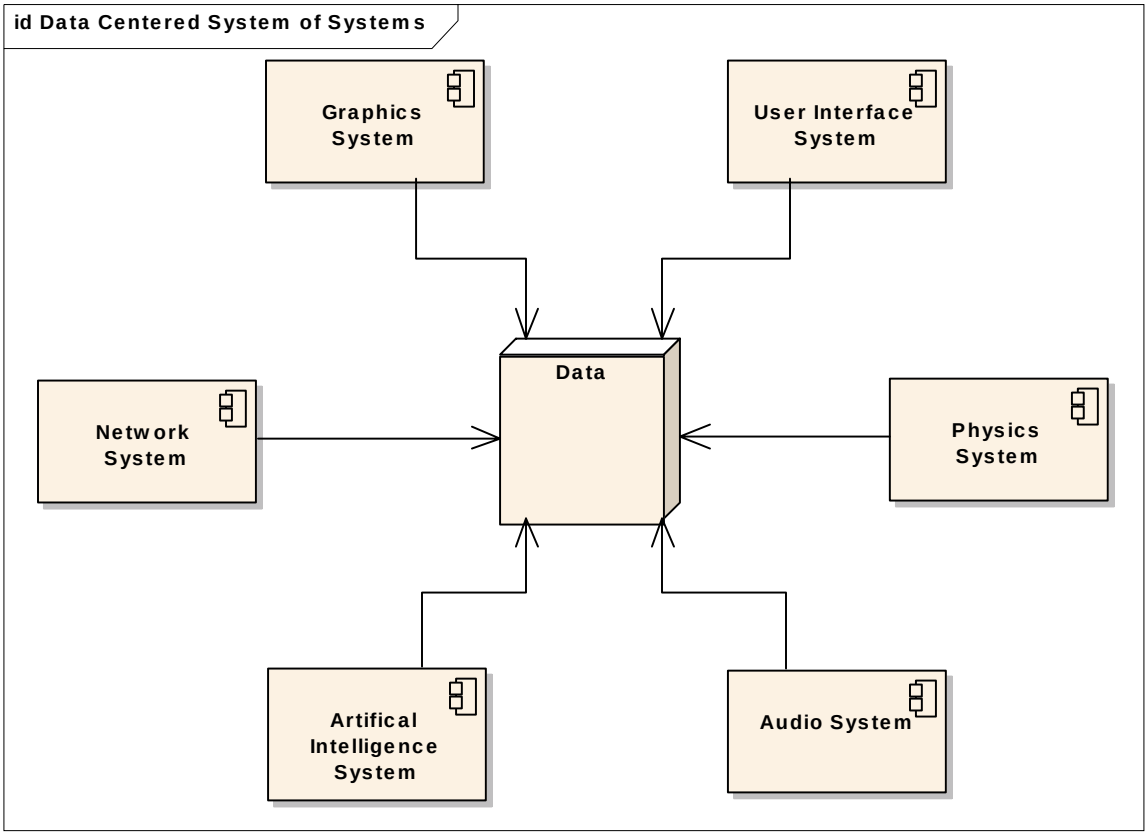

Figure 20 - Data Centered System of Systems

One nice feature of the design shown above is the minimization on dependencies. Sub-systems no longer depend on each other, they can only work with data and by working with the same data they are working with each other. Such decoupling should mean that any sub-system could potentially be replaced or modified without breaking any of the other components.

The concept presented in Figure 19 is definitely an interesting approach but it has one fatal flaw that any gamer would immediately notice – speed. Games are expected to run at very fast speeds, any thing less and the product would be summarily dismissed as a failure. The above design would suggest that each system processes on the whole of the

data. In an era where the data content of a single game can span multiple CDs, this is obviously not a feasible approach.

This brings us to the second major design decision – selective data processing. Taking a page from existing game development knowledge, we know the mathematically complex and time consuming graphics system doesn't need to process all data objects in the game, only the objects in the player's immediate area. In fact just about every sub-system could benefit from some sort of spatial data organization or scene management. By moving from a simple data store to a complete data management system we can move back closer to the performance of existing game architectures (See Figure 20).

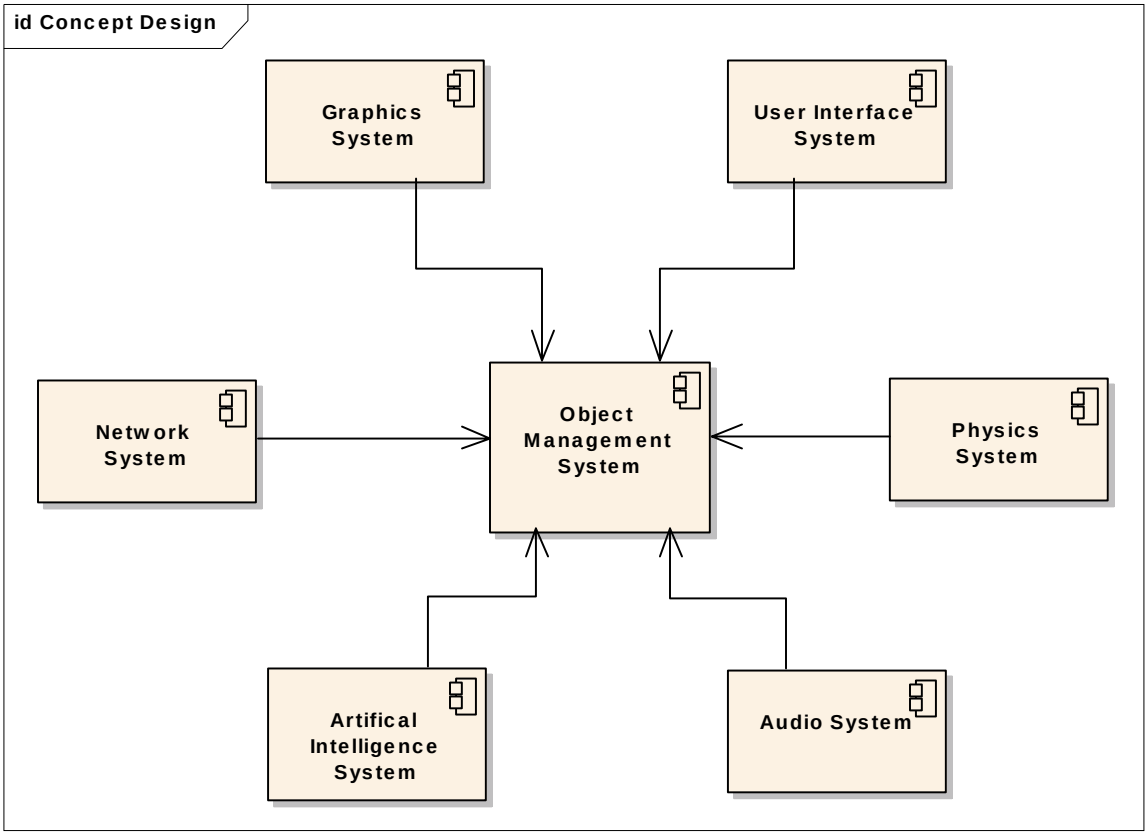

Figure 21- Intelligent Data System Centered System of Systems

4.2 Architecture – System Communication

As shown in the analysis phase, system communication can be performed a variety of different ways. Continuing with the system of systems idea, each domain-specific component working with the object management component is actually an independent system (see Figure 21 below). Since a complete system is composed of only two components and single connection, a direct connection or function calls is acceptable. Allowing direct communications is actually preferable considering the performance constraints that exist in the games domain.

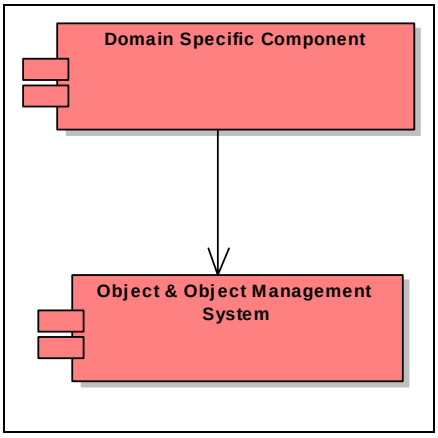

Figure 22 – System Defined as a Domain-specific Component & the Object Component

Allowing direct interface communication between a domain-specific component and the object management system fortunately doesn't have much impact in terms of flexibility and expandability. All domain-specific components require virtually the same types of interactions with the data. They only require lists of objects to operate on, and the ability to read and write to those objects. As long as the design supports those kinds of interactions, the system should remain easily modifiable and expandable.

4.3 Architecture – Synchronization

Software architects might immediately notice that this architecture doesn't support much in the way of component synchronization. There is no direct communication between domain-specific systems so there is no immediate way for one domain-specific component to tell another that it has modified data the other was using. In some application domains this could be a very serious problem, but keep in mind this architecture is for games. If for a single tick an object gets drawn even though the AI system determined that it was killed, a player isn't likely to even notice let alone care.

Synchronization does exist, but it is performed at the system level rather than the object level. Unlike architectures where synchronization exists at the object level, it is no longer enough to "tick" each object in the relative scene and trust that the object will be drawn, act out its behavior, make sound, etc. Now each system must execute on the data in turn. A master system must tell a component to operate on all the relevant data and then signal the next component to do the same. (see Figure 22 below).

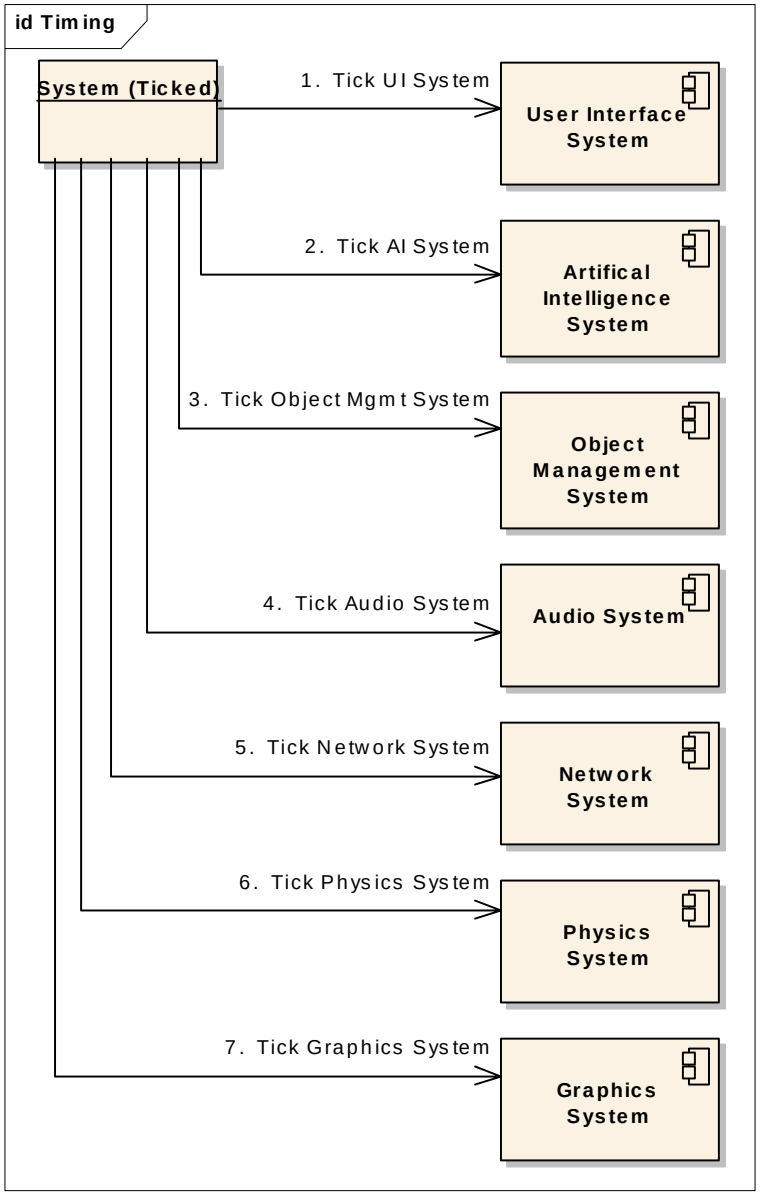

Figure 23 - Ticking the Game System of Systems

4.4 Architecture – Distributed Synchronization

It is important to note that the architecture assumes each component is operating on the same computer. This method of synchronization is not plausible if the domain-

specific systems resided on different platforms. This does not mean, however, that distributed games cannot be developed using this architecture. In fact creating a networked game is a simple expansion of adding a networking component to the local system.

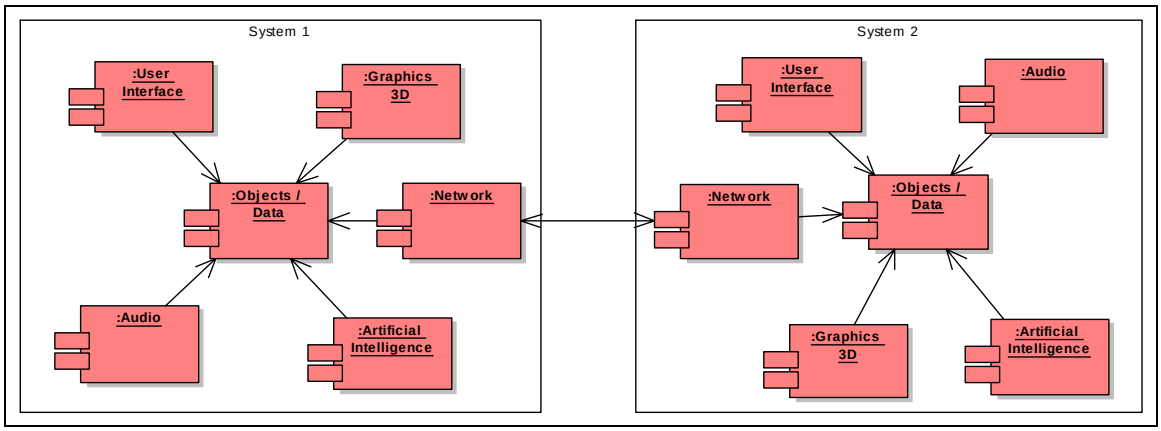

Figure 24 – Example Peer to Peer Networked Game

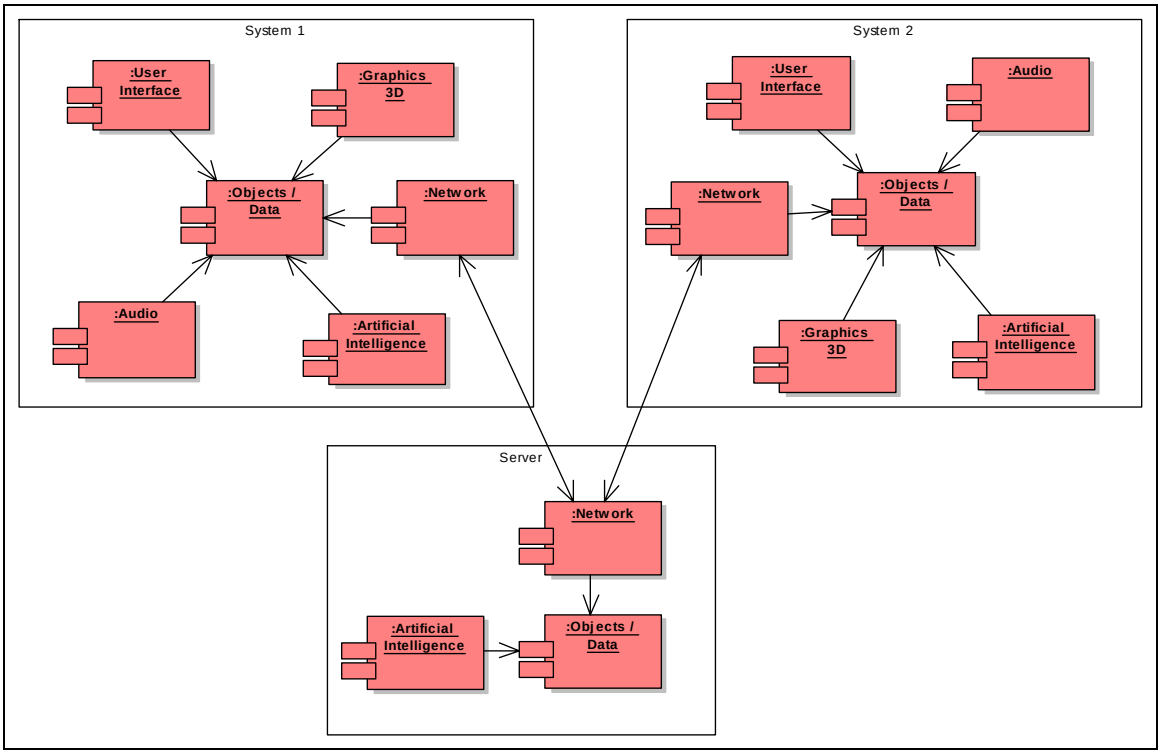

Figure 25 -Example Client Server Networked Game

As you can see in Figures 23 & 24 above, the architecture is capable of supporting the most common networking models. Game developers are free to create their own method of game synchronization. You'll notice that the server system in diagram 24 above has not only a network component, but an AI system as well. This was added because games often use estimation logic in the server to keep clients reasonable synchronized due to the fact that different clients have different quality of network connections.

4.5 Architectural Features / Architectural Requirements

The goal of the proposed architecture was designed to meet the requirements stated earlier in Chapter One. While at this stage of the thesis it has not been proven that the proposed architecture will meet all of the requirements, it does look promising. Actually validating the architecture will be presented in later chapters.

4.5.1 Support for COTS-Based Development

The proposed architecture seems to support COTS-based development very well. Functionality is separated and integration is reduced to a simple logical interface. As long as the COTS component can request objects to process, and is capable of operating on the object data, game systems should have little difficulty integrating external systems.

4.5.2 Better Knowledge Localization

At this point it isn't immediately discernable whether the architecture supports better knowledge localization than other approaches. In one sense it does because the only cross component communication is that of requesting objects to operate on, thus removing the need for game developers to learn complex domain-specific APIs. On the other hand, the object system must support domain-specific data in order for the components to operate properly. We shall see a little later on that this concern can be mitigated in the design phase.

4.5.3 System Flexibility / Modifiability

The architecture appears to be flexible. The data-centered topology allows for any type of system to operate on the data, so seemingly any type of game could potentially be

created. It is the developer's choice of components and their functionality that determines the type of game being produced. In later chapters this thesis will attempt to more solidly prove that the proposed architecture supports this requirement.

4.5.4 System Expandability / Maintainability

The figures above represent potential game systems comprised of several subsystems. While there is nothing wrong with the potential game design, the diagrams don't show one of the architecture's greatest strength – expandability. The figures above show one artificial intelligence system executing in a game, but there is no reason there can't be more. Consider the possibility of having one AI system that determines unit strategy, while another performs path-finding from one location to another across a map (see Figure 25 below).

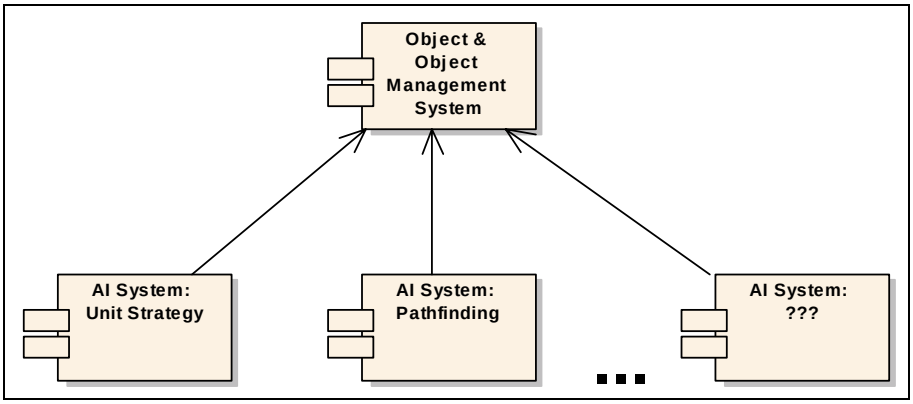

Figure 26- Potential Design using many AI Systems

The architecture readily supports the ability for game designers to use any type and number of subsystems they choose. This also promotes COTS development and re-use,

since developers can easily re-use some of the more general subsystems, like path-finding AI, across multiple games.

4.6 A Simple Design

In order to verify the architecture is even feasible, a very simple design will be created. The design is not intended to be the official starting point for games to begin development from. It is simply meant to ensure that it is possible to create a game system using the proposed architecture. Future research will include building better designs using this architecture, but for this thesis simplicity is the only requirement.

4.6.1 Potential Design: System Communication / Interaction

The architecture defines the topology of any design components, so the fist step is to determine how the individual systems will collaborate via the object management system to form a cohesive game system. Using the proposed data-centered approach there are two kinds of interaction that are of interest. First is the interaction to attach an external system to the object management system. It should be generic enough that any number and type of component should be able to attach in a similar fashion. The second important interaction is the actual reads and writes that take place between the object system and an external system. The method of interaction must be generic enough that all subsystems can use it, and flexible enough to support the different kinds of interactions domain-specific components will need.

4.6.2 Potential Design Cont.: Attaching Systems at Compile Time

The ability to attach any type and any number of systems to the object management system is critical to the architectural requirements for flexibility and expandability. Because the details of the design are only interesting from an architectural feasibility standpoint, the simplest design was taken and systems will connect to the object component via semi-standardized interfaces (see Figures 26 & 27 below).

The approach below is definitely not the best but it does work. Domain-specific systems require the object management system to implement a specific interface. The domain-specific system will then communicate with the object system via that interface. It is system expandable at compile time by having the object system implement a new domain-specific interface and having the game system of systems attach the new domain-specific system. This design is not too bad if the interface the object system is required to implement is kept simple, which it will be.

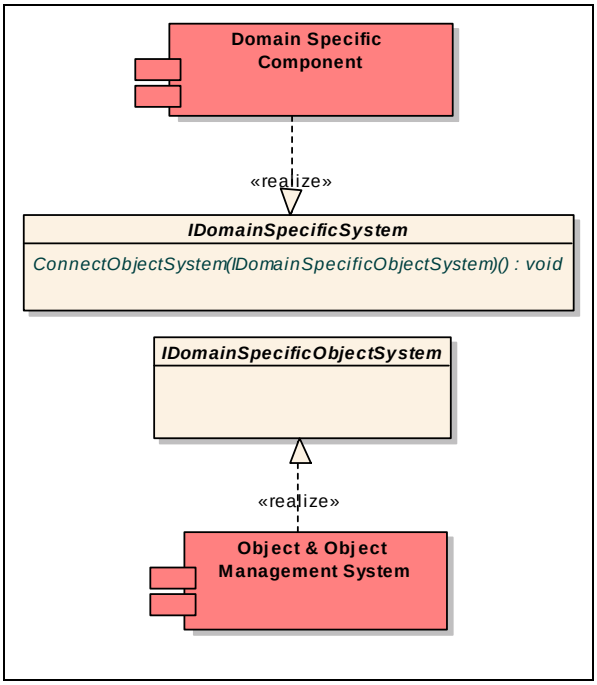

Figure 27 – Interfaces Required to Connect Domain-specific Component to the Object Management Component

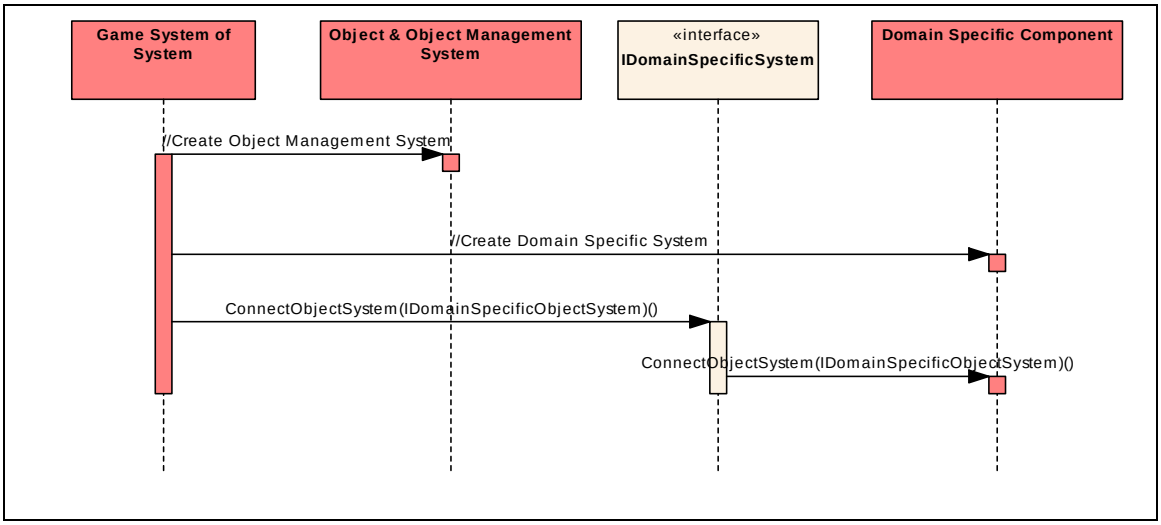

Figure 28 – Example Sequence of Connecting a Domain-specific Component to the Object Management Component

4.6.3 Potential Design Cont.: System Communication

Continuing with the design presented above we need a design that provides a simple and generic way for the domain-specific systems to interact with the object management system. Fortunately the interactions required is simply one of getting data objects for domain-specific processing. I've found that a view/object-list interface provides generic enough access, and is flexible enough to meet the data access needs of the domain sub-systems (see Figures 28 & 29 below).

Essentially each domain-specific system needs to request object lists or iterators of objects to process. The view provides constraints and a context for the object list. For example, a graphics engine requires lists objects that should be drawn. In order to do this the graphics engine might receive a view that provides context stating these objects should take up the whole screen, and then provides the list of visible objects to draw. It might also receive a small view that states to draw the contained objects in the upper left hand corner, and provides a list of GUI objects to draw.

An AI system, on the other hand, might only require a single view that allows the AI system access to all the objects within 100-meter radius of the player, or perhaps a simple list of computer controlled creatures. So while both the graphics and AI systems require different lists of objects, the view / object-list approach is flexible enough to meet the needs of both.

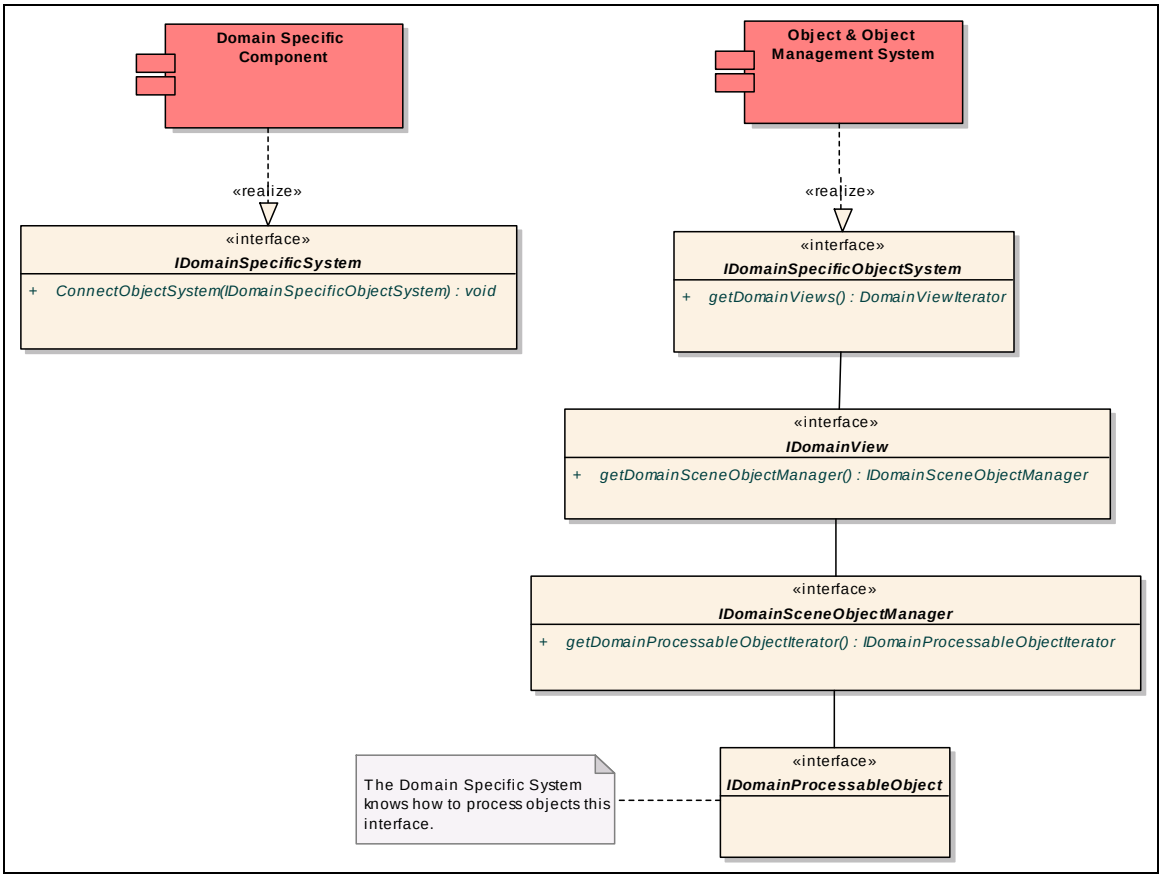

Figure 29 – Interfaces Required for Domain-specific System To Request Objects to Process

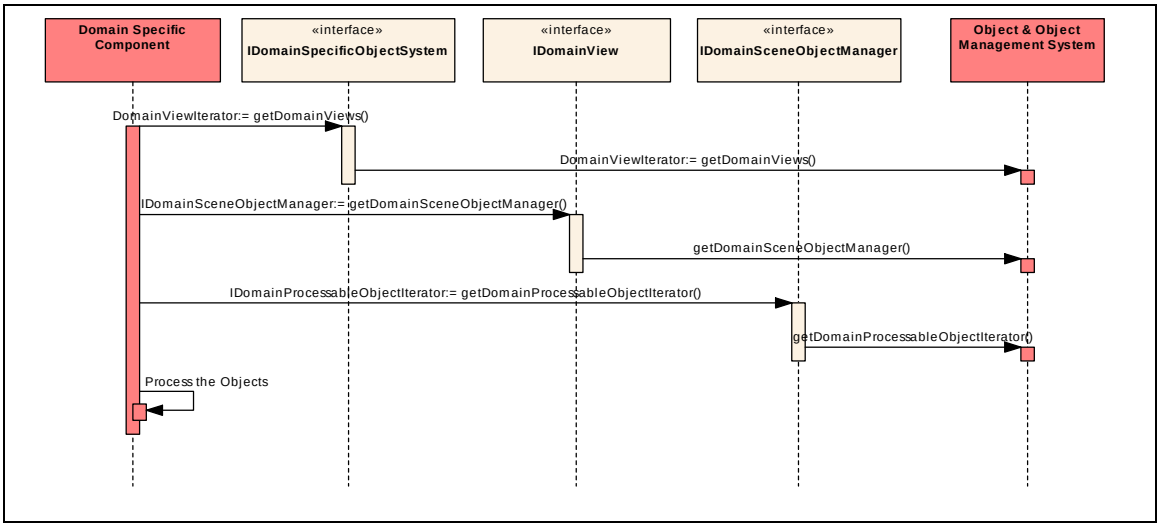

Figure 30 – Example Sequence of a Domain-specific System Requesting Objects to Process

4.6.4 Potential Design Cont.: Observer Pattern to Achieve Localization of Domain Knowledge

One of the initial architectural requirements is to support domain knowledge localization. For example, the graphics system contains a great deal of domain data like mesh and animation structures that is directly related to the game objects in the object management system. The designer of the object management system and even the game specific objects should not need to know about those domain-specific details. A game developer should care that a game object is “attacking”, not necessarily that a specific graphics engine, with specific class objects is being used to represent the attack visually.

One possible solution to this problem is the observer design pattern (Bass et al). If objects in the object management system had the generic capability to attach and retrieve observer objects, domain-specific systems could attach domain-specific data for processing without the object system needing to understand the data. Figures 30 and 31

below show a simple example of how a simple object can be expanded to contain domain-specific data without the game object creator needing to understand the specific domain. So for example, the graphics engine could attach an object that contains the 3D mesh, a skeleton, material information etc. as an attached object, and the game object need never know it contains graphics specific information.

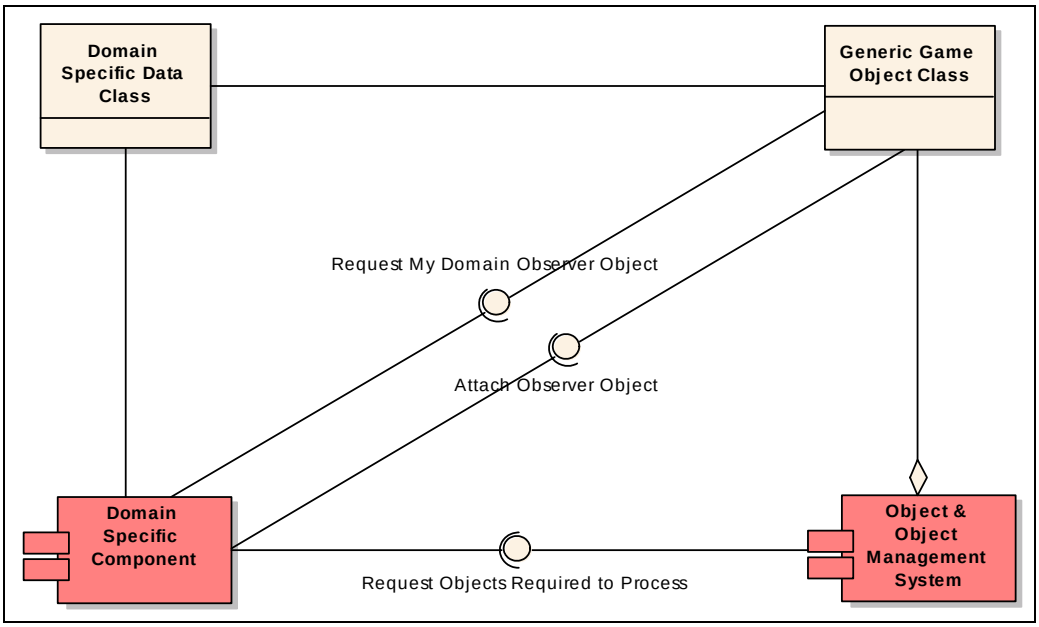

Figure 31-Potential Design using a Domain Observer Object

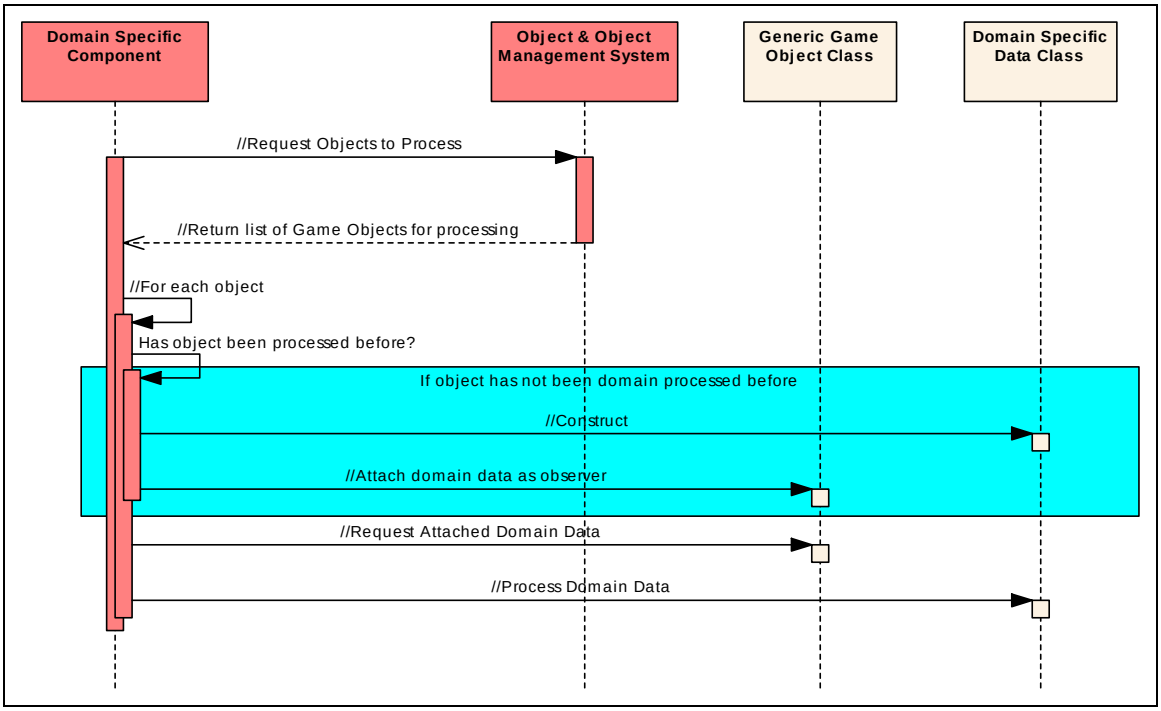

Figure 32-Potential Sequence using a Domain Observer Object

5 ARCHITECTURE VALIDATION

The next step in developing the architecture is to verify to a reasonable degree that the architecture supports the functionality for which it was intended. The first approach is to apply the architecture and take the reference games past the functional level to the design level. This should prove to a fair degree of certainty that the architecture still supports the different game functionality.

The next validation technique used in this thesis is to build a prototype system using the proposed architecture and confirm that the original goals and requirements have been met. Obviously building a commercial quality game like Starcraft™ or Unreal Tournament™ are beyond the scope of this thesis, but building a prototype that offers a subset of functionality can be created in a reasonable amount of time. The prototype system should demonstrate each of the original architectural requirements.

5.1 Taking the Reference Games to the Design Level

5.1.1 Applying the Design

The first step in proving the architecture is sound, is proving the architecture can at least support the functionality it was designed for. Carrying the original game analysis to the design level should show that the games could have been built using this architecture. This step will not, however, show if the architecture would work well for the given game application. The architectural qualities will be left for the prototype to demonstrate.

Before we can carry on to software design, the original analysis artifacts must be reviewed to see how the proposed architecture affects our understanding of the game. Going back to the “Play Starcraft” use case diagram in Figure 9, there is one major

problem that needs to be solved. While it appears to capture the activities a human player can perform, it is still incomplete for trying to understand how games really play. The reason is the timing model for games is very different than the typical software application.

Most literature proposes use-cases to capture the interactions between actors external to the system and the system being developed. Games are slightly different, however, in that the player does not initiate all forms of interactions. For example, if a player starts a game of Starcraft™ and never enters another command, the game will still play. The computer AI will process strategies, units will move and behave, and ultimately the game will continue without the player. By bending the rules slightly and treating the clock as an actor, the transactional use-case approach should still be sufficient for capturing the functional requirements in our design.

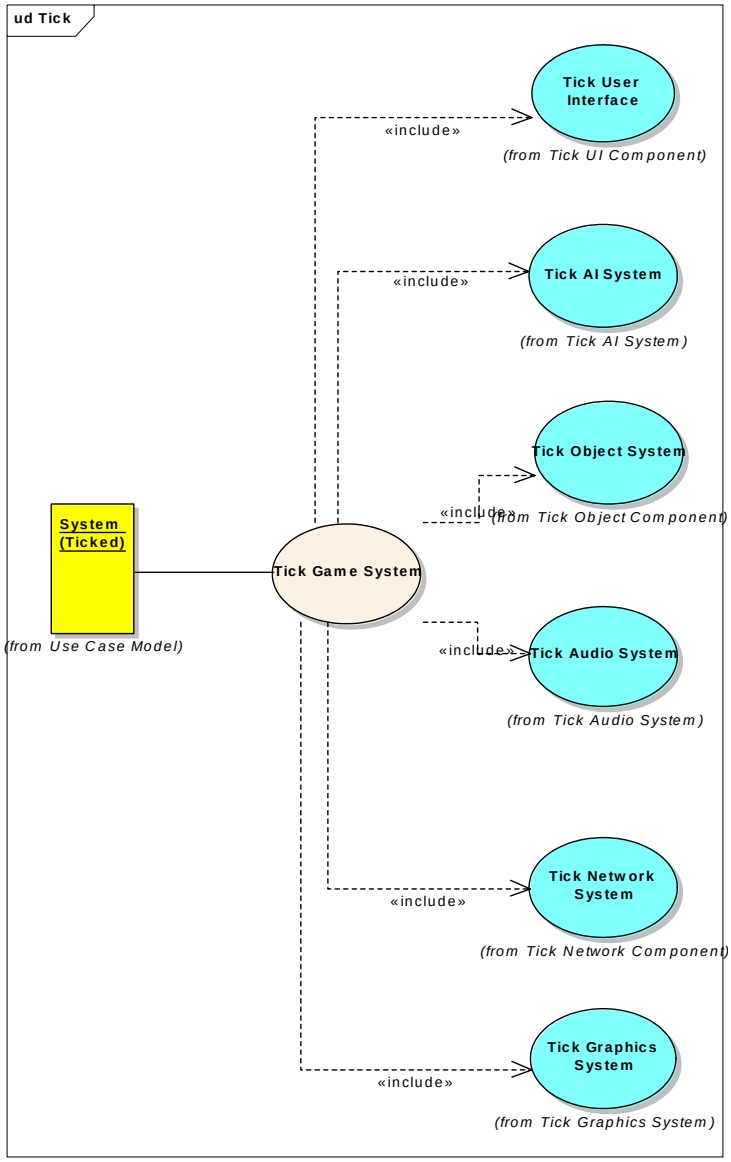

Figure 33 - Tick Game System Use Case

By looking at the original use-case list in Figure 9 in terms of how they would break out in terms of the timing use-cases in Figure 32, we can start to understand how the logical subsystems might implement the game functionality. From here we can begin to break down the use case and assign portions of it to the various sub-systems. Figure 33

below shows a possible use-case breakdown of the “Tick Graphics System” use-case for the game Starcraft™.

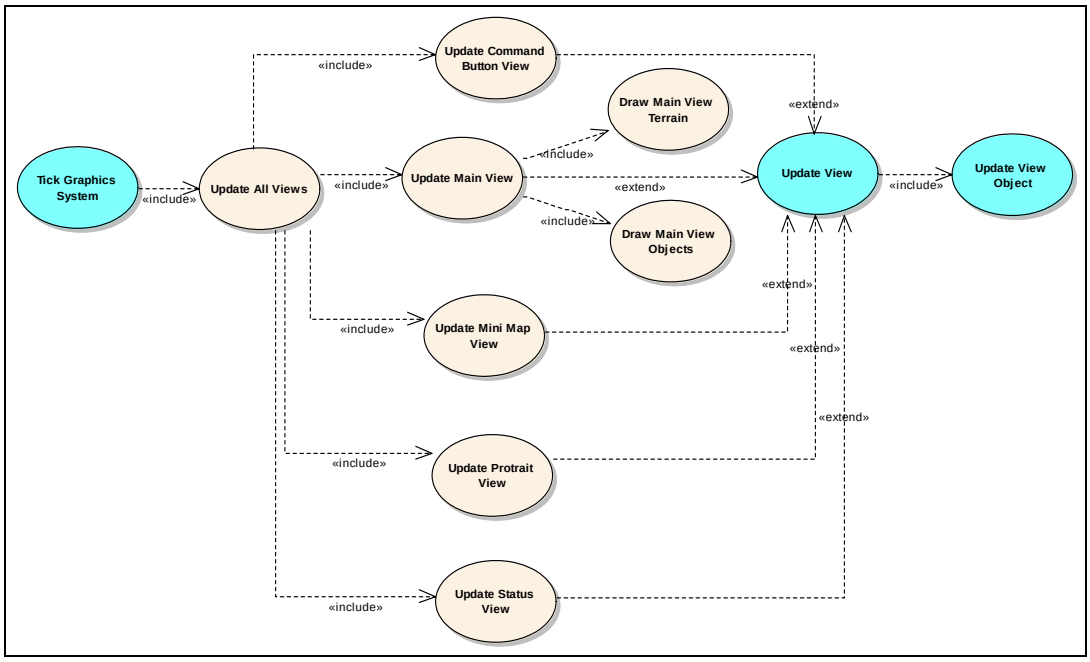

Figure 34 – Tick Graphics System

Selected use cases are then further expanded similarly to what was done during the analysis phase, only this time the simple design is used. Use cases are driven down to the system interactions, which are then further driven down into the actual interfaces involved (see Figures 34 and 35 below). At the end of this we have not only validated that the analyzed games could be like be built on the proposed architecture, but we have further defined the interfaces which will be useful for the prototype effort. For the detailed designs of Starcraft™ and Unreal Tournament™ see appendix A.

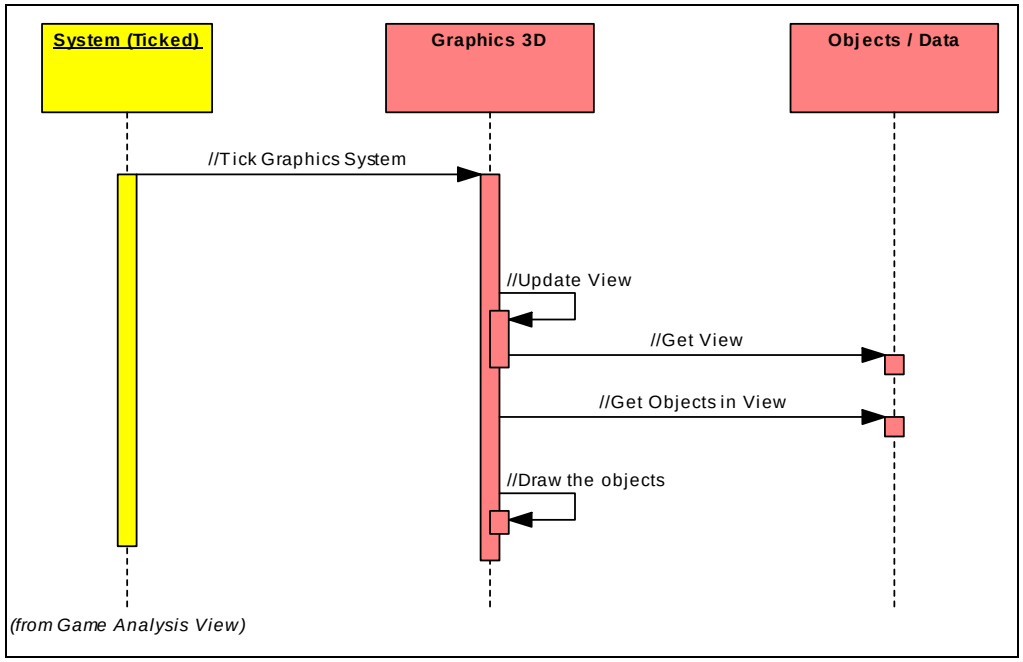

Figure 35 – Update View Component Sequence

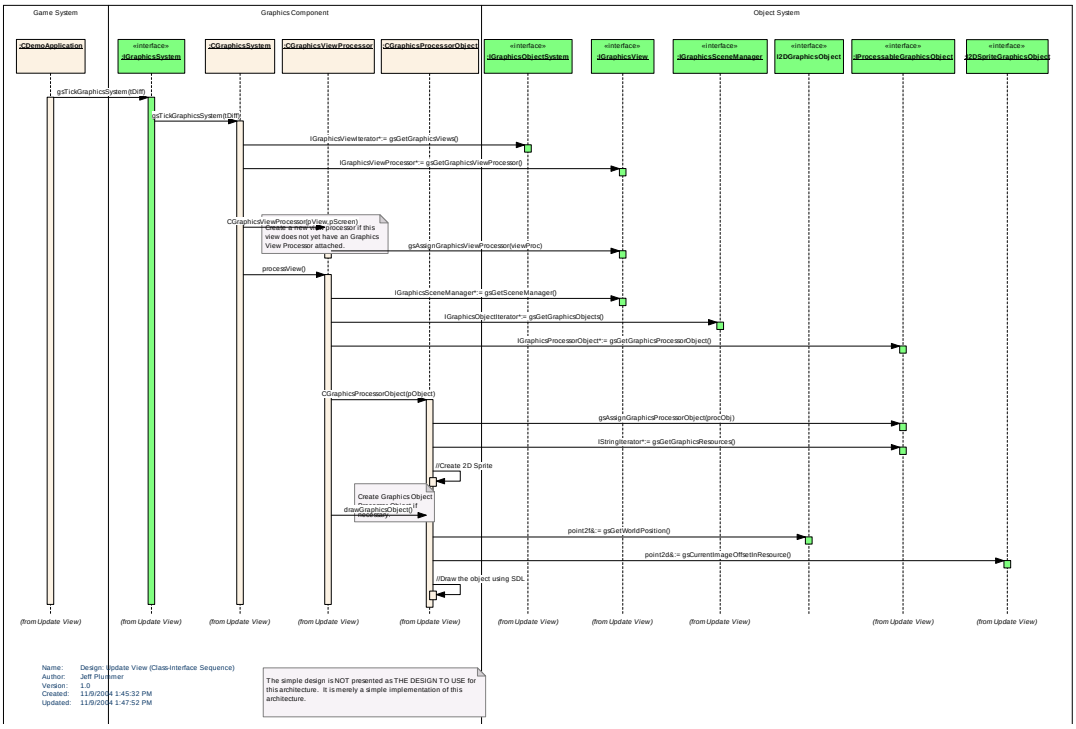

Figure 36 – Update View – Classes and Interfaces

5.1.2 Evaluating the results of applying the design

Before we analyze the results of this exercise it's important to truly understand the goals. Moving the selected games down to a design level was performed to verify that these types of games could be built using this architecture. The resulting artifacts do not give any insight as to how well the architecture fits the games domain. The artifacts also are design dependent, so the level of complexity in these representations is more a reflection of the quality (or lack thereof) of the design, and not the architecture.

Overall it would appear that the architecture can support the two selected games, and therefore arguably supports many types of games. Using the proposed architecture it was possible to design the kinds of functionality required for both games. The idea of individual systems operating on the same data was a bit of a paradigm shift from what is

commonly seen in game development literature, but the shift was not so large to make it a difficult transition.

5.2 Developing a Prototype

In an industry where changing people's perceptions of software engineering is so difficult, a paper analysis of the architecture is not likely to change anyone's development habits. A tangible prototype that can demonstrate the architectural qualities in a game-like application is far more likely to have an impact. A prototype will also more concretely prove the quality attributes this architecture purports to have.

5.2.1 Prototype High Level Design

An effective prototype for this thesis needs to meet certain criteria. First the prototype must have game-like functionality. It should demonstrate some of the same kinds of capabilities that exist in games. Second it should demonstrate all of the architectural requirements stated in Chapter One of this thesis. Lastly, even though performance was not one of the architectural requirements, the prototype should execute at speeds reasonable to games. An application that meets such criteria should be able to answer a great many of the questions likely to arise from people familiar with game development.

5.2.1.1 Component Selection

The first task in developing the prototype is deciding which systems to model and build. In order to best demonstrate the architecture's support of our defined requirements, most notably flexibility and expandability, only a few domains will be

developed. AI and graphics seem the logical choice and should offer ample opportunity to flex and expand.

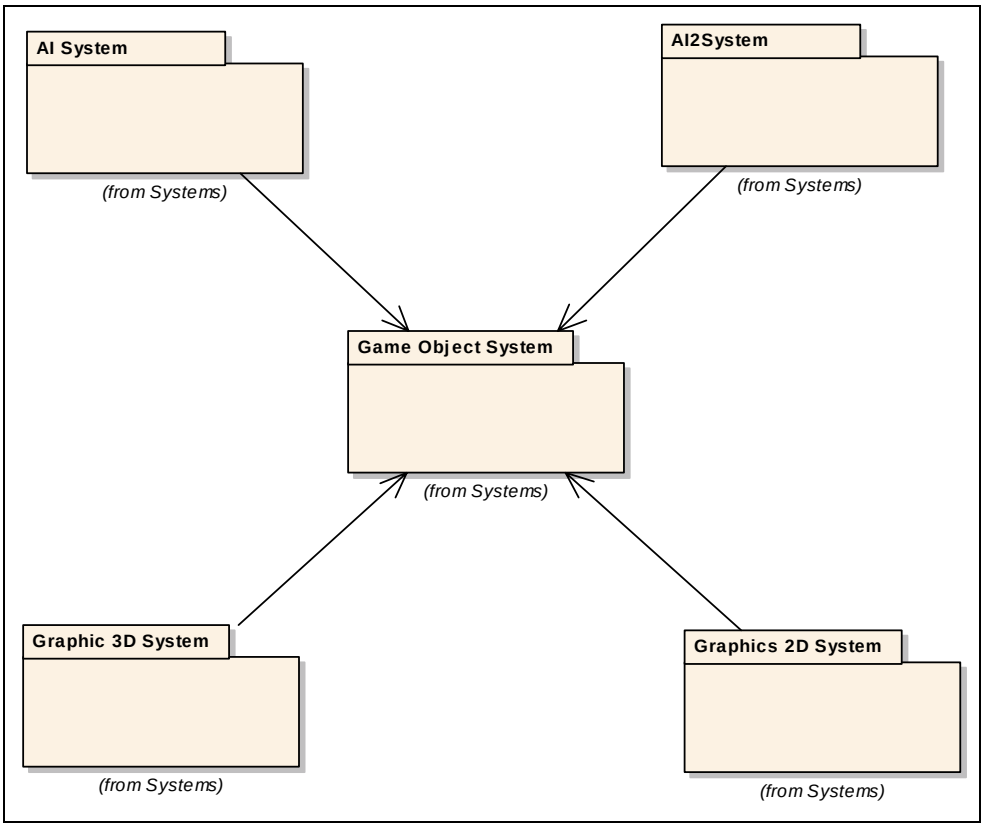

Figure 37 – Prototype Subsystems

Figure 36 above shows the logical systems that will be built for the prototype. The prototype should show flexibility in the way the "game" can be assembled using any combination of these components. It should also demonstrate expandability because moving from a 2D graphics system to a 3D graphics system is a logical upgrade.

Game Object System - This component acts as the data store that all other systems will interact with. It is also responsible for organizing the list of objects the domain systems will operate on.

AI System	- This is an extremely trivial intelligence system that will tell objects to move around.
AI2System	- This is another trivial intelligence system that tells objects to rotate.
Graphics 2D System	- A 2D graphics system that renders sprite objects.
Graphics 3D System	- A 3D graphics system that renders 3D objects.

5.2.1.2 The Object Data

The next step is to identify the object data that each system uses to operate on. Figure 37 below shows the object data required for this prototype. This example design also shows how a single data set can be re-used. For example, when the 2D graphics system requests an object position as a point2d (structure of two integers) the object can simple simply return integer typecasts of the x & y aspects of its point3f (structure of 3 floats) location. So in essence when the AI system modifies the position data, it's modifying the position data that all the components use.

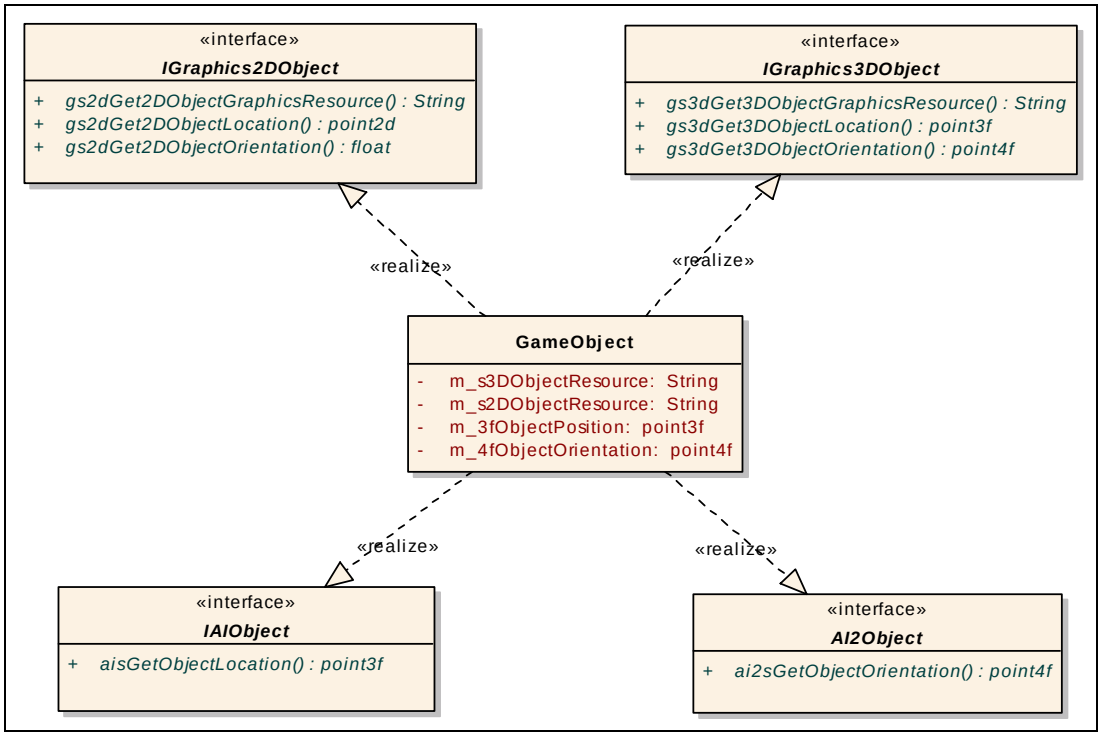

Figure 38 – Analysis of Object Data Required

While this prototype diagram suggests that the object implements interfaces from the various systems, that is a design choice not an architectural requirement. Other designs may use other (possibly better) methods of interacting with the data in the object. Our purpose here is simply to ensure that we understand what data the attaching systems will manipulate.

5.2.2 Prototype Detailed Design

The prototype system will follow the design proposed earlier in Chapter 4. We will soon see it is not the best possible design, but it is simple to understand and adequate for our uses. Components will request views (a view is really just a list of objects as well as

some context information), and then process the objects in that view. So for example when a graphics component requests a view, the object system would provide a view that contains the list of likely visible objects.

The design also uses domain-specific observer objects to be attached to the data objects. This allows the domain-specific system to attach domain data to the object without the object component requiring any kind of special understanding of the domain data. As stated before, this design feature was added to provide for knowledge localization.

5.2.2.1 Component Interfaces

The simple design uses interfaces to facilitate communication between the domain-specific system and the object management system. Each domain-specific component will present two kinds of interfaces. One set of interfaces the domain-specific system will implement and present to the game maker / object management system. At its simplest, these interfaces are ONLY for connecting the object system to the domain-specific system, thus keeping the complexities of the domain hidden entirely from the developer. The other set of interfaces are to allow the domain-specific component to use the object system. At its simplest, these interfaces are ONLY for requesting views and access to certain object attributes (see Figure 38 below).

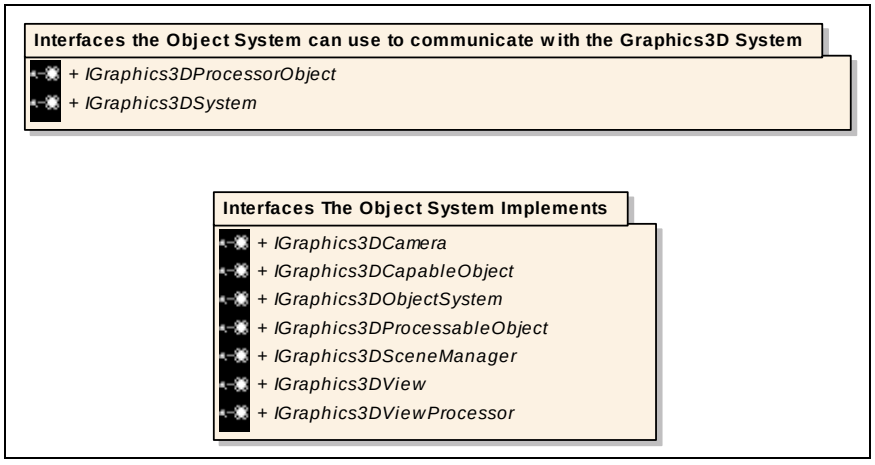

Figure 39 - Example: Graphics3D System Interfaces

In keeping the goal of knowledge localization, the interfaces the domain system presents to the game maker are trivial. In the example below in Figure 39, the IGraphics3Dsystem interface provides mechanisms for the game maker to attach an object system, configure, and "tick" the system. The IGraphics3DprocessorObject and IGraphics3DViewProcessor are the interfaces to allow the domain-specific system to attach observers to a data object and view respectively. Such interfaces could also potentially provide the object and view access to domain-specific functionality, allowing game developer to play with the nuts and bolts of the domain-specific system. They are left empty for this demo because one goal of this demo is to demonstrate that game systems can be assembled without the game developer using any of the domain-specific functionality. Virtually all domain-specific systems will present a nearly identical set of interfaces using this design.

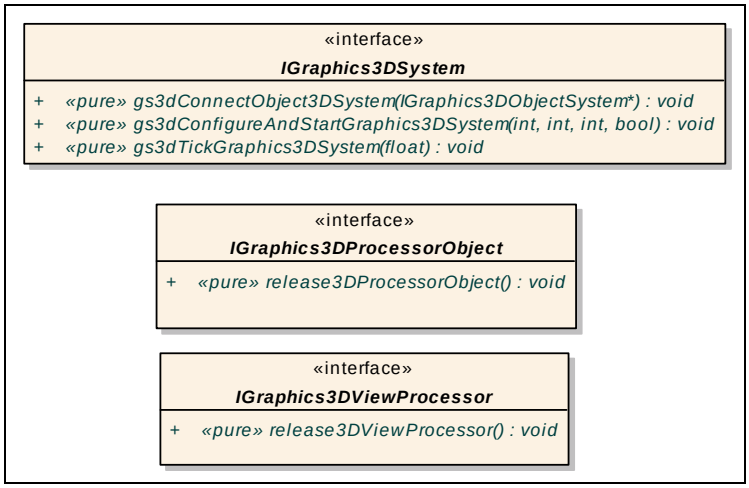

Figure 40 – Interfaces Into the Graphics 3D System

The interfaces the domain-specific system places on the object system to implement are equally trivial. The IGraphics3DObjectSystem interface allows the graphics system to retrieve views. The IGraphics3DView provides access to the objects that should be considered for drawing, as well as context information like the camera and view rectangle. Finally the IGraphics3DProcessableObject interface allows the graphics system to attach an observer, and allows access to the data the graphics system is interested in. And just as before, virtually all domain-specific systems can use a virtually identical set of interfaces using this design.

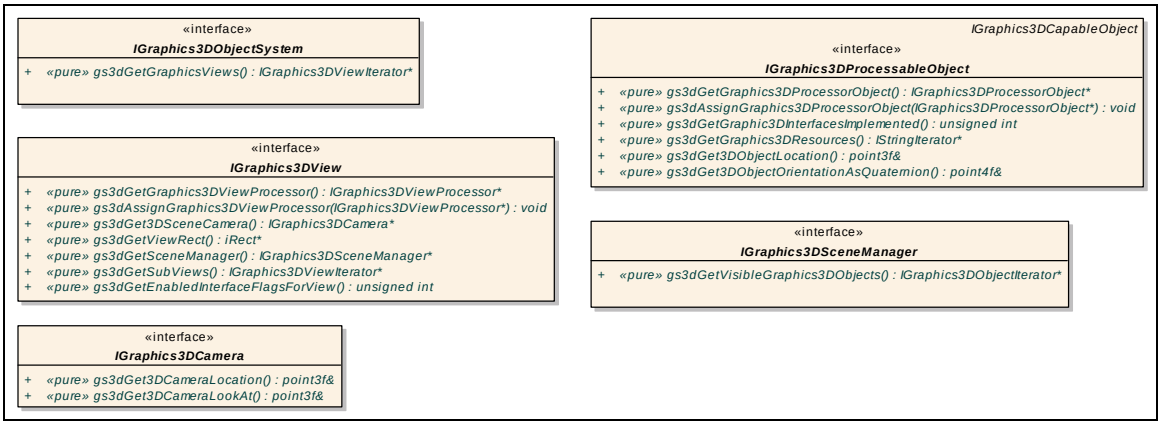

Figure 41 – Interfaces the Object and Object Management System Must Implement in order for the Graphics 3D Component to Use it.

5.2.2.2 Domain-specific System – Object System Interactions

This simple design requires only two types of system interaction. The first occurs at system creation time where the domain-specific system is connected to the object system. The second is the interaction that occurs when you "tick" the domain-specific system. The combination of "ticking" all the domain systems should result in the game system.

5.2.2.2.1 Connecting Domain System to the Object System

The simple design used in this prototype connects the individual systems via interfaces. This extremely simple interaction provides the domain-specific component an interface to use to communicate with the object system. Figure 41 below shows an example of how the system simply passes a reference to the object component to the graphics 3D component. Once the domain-specific component has the interface to the data it can process the data via the "tick" command.

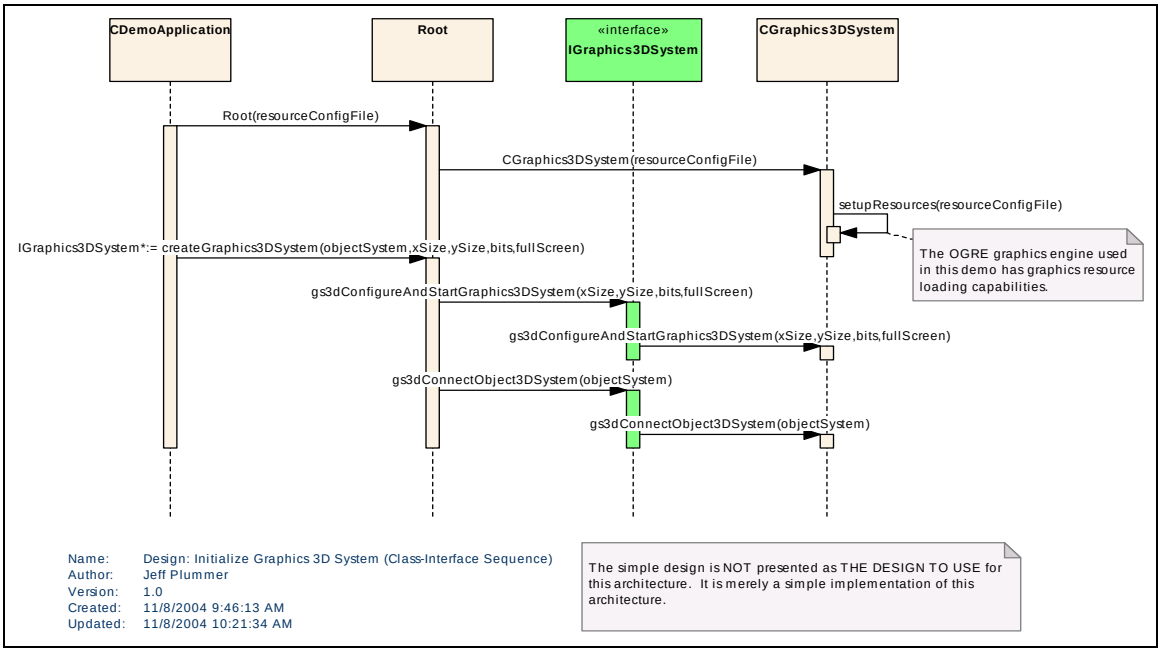

Figure 42 – Connecting the Object Component to the Graphics3D Component

5.2.2.2.2 "Ticking" the Domain-specific System

"Ticking" the domain-specific system is where the real work is done. The system tells the domain-specific component to synchronize and process the data in the object management component. To do this, the domain-specific component will request views and object lists to process, perform domain-specific functionality, and update the data in the object management component. The simple prototype design uses interfaces to perform this and an example can be seen below in Figure 42. See Appendix B for the complete prototype design.

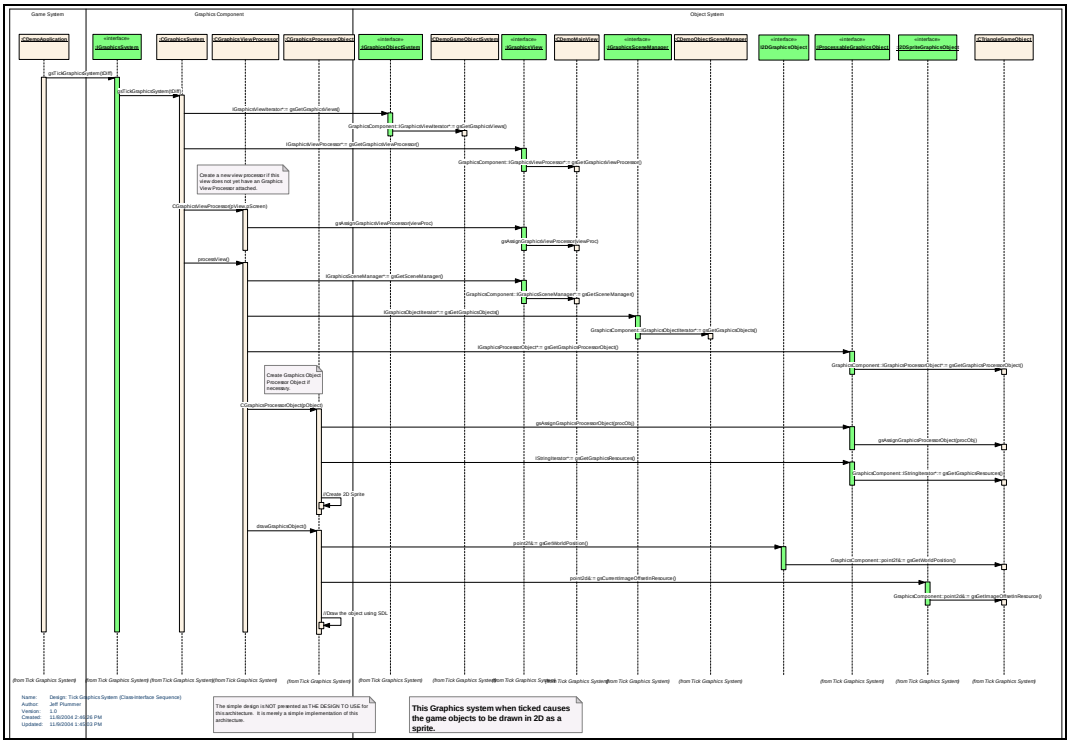

Figure 43 – Prototype Sequence: Tick Graphics2D System

5.2.3 Prototype Evaluation

Designing and building the prototype was a much larger task than originally anticipated, but it was definitely worth the effort. First and foremost it proved the architecture definitely has potential. The prototype not only demonstrated the original quality requirements but it uncovered several issues that had not previously been considered.

In terms of flexibility, the prototype demonstrated the ability to attach and remove AI components quickly and easily, and resulted in notably different “game” behavior. For expandability, the “game” could quickly move from 2D to 3D by attaching a new graphics engine. In terms of domain knowledge localization the game specific objects

had no domain knowledge about the components that would use them. The game developer merely had to implement simple data access interfaces, and domain-specific data was hidden as an attached observer object. Lastly, the prototype did prove the architecture supports COTS based development. The domain-specific components were separate by some very simple interfaces and required no understanding of the inner workings of other systems.

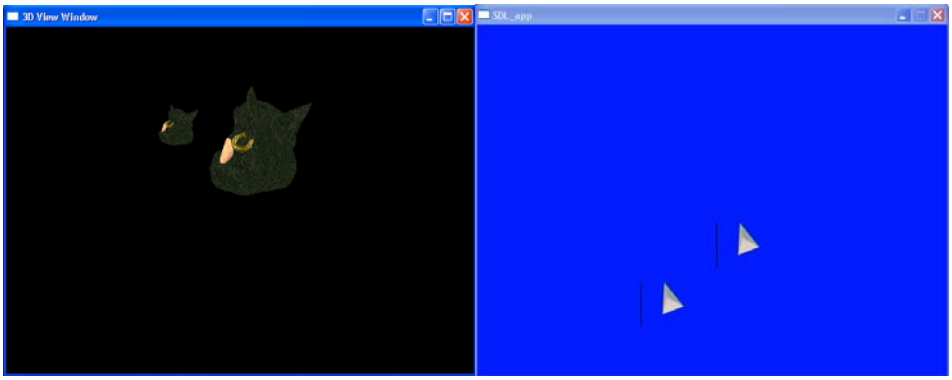

Figure 44 – Screenshot1 from Prototype

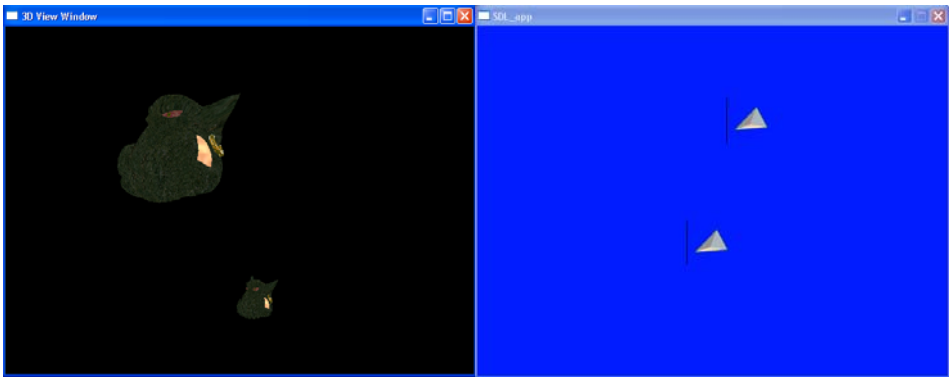

Figure 45 - Screenshot 2 from Prototype

6 RESULTS

As technology advances and consumers demand the latest features, electronic games will be required to continue to grow in terms of size and complexity. In order for development houses to keep costs low, certain realities must be faced. Games can no longer be coded entirely from scratch. The total cost in terms of time and resources will soon reach a point making games an infeasible venture. This thesis has proposed an architectural solution that could help mitigate this problem by moving games to a common COTS architecture. By allowing developers to "assemble" the game framework in a flexible manner from technology components, game makers can spend more of their time focusing on the more game specific aspects.

6.1 Summary

Electronic games are making incredible advances in terms of technology and complexities. Unfortunately almost all-available literature on the subject of games only tends to keep pace with the technology advances, leaving developers to devise their own solutions for managing the complexities. The emerging field of software architecture is an area of research that has been shown to drastically impact the development of large complex systems. By using the knowledge found in software architecture and applying it to the games domain, we can begin to fill the gap that is left by current literature.

In order to design a quality domain-specific architecture, a solid understanding of the games domain needed to be acquired. Such insight was achieved by analyzing existing games using standards software engineering practices. The resulting artifacts presented a quality understanding of the kinds of functionalities and interactions that can occur in modern day games. Then came the task of finding architectural styles that offer a nice fit

for those kinds of interactions. The resulting research also sparked a profound interest in the system of systems philosophy. Weighing the pros and cons for each style as applied to the domain resulted in a solid knowledge base for designing an architecture that could meet the needs of the games domain.

The next phase was to actually use all the acquired knowledge and design an architecture for the games domain. The proposed architecture was defined as using a data-centered topology, a direct method invocation for communication, and using a system "tick" for synchronization. The game system emerges as a result of multiple independent systems working on the same data set. A simple design was then created that could be used for the remaining analysis.

After building a simple design it was time to begin analyzing the architecture. The first method of analysis was to carry the selected games for analysis down to the design level. This should verify if the architecture is at least capable of support the analyzed games and by implication capable of supporting many other types of games. In order to determine how well the architecture would support games the simple design was used in a prototype system. The prototype was then used to demonstrate the quality attributes of the architecture.

6.2 Conclusions – Meeting The Architectural Requirements

The proposed architecture definitely shows promise for use in the games domain. It appears to support the functionality required in a diverse set of electronic games, and it appears to support them quite well. The architecture allows for a great deal of flexibility and expandability by supporting any number of a wide variety of domain-specific

systems. The architecture also supports the COTS based development approach, and the easy integration of those components. All in all the architecture seems to offer a great deal of benefits over the more ad-hoc, tightly coupled designs used today.

6.2.1 Support COTS-Based Development

The proposed architecture promotes COTS-Based development by eliminating domain-specific component communication. Domain-specific components can only communicate with the object component, and together form an independent system that exists completely independent of any other system.

This architectural requirement was verified during the development and assembly of the prototype. During development, components required only a simple object system to create a fully demonstrable and testable system. During prototype assembly, the prototype components can be added and removed at will, demonstrating the complete independence of the individual components. The domain-specific components also integrate in a near identical fashion simplifying their integration.

6.2.2 Better Knowledge Localization

The proposed architecture also supports the ability to localize domain knowledge away from the game developer. By eliminating, or at least greatly reducing, the amount of domain knowledge a game developer must understand in order to use a game component properly, we are effectively giving the developer more time to focus on the game specific aspects of the code.

While such a feature is not immediately inherent in the architecture, it is possible to add this at the design level. This thesis expanded the architecture to a design that used

the observer design pattern to attach domain-specific data to a game object. In the prototype, the game specific objects have very little domain-specific data. For example, the only 3D Graphics specific piece of data the object component has is a string that says what 3D graphics resource to use. All the underlying data that is needed to render that resource is attached as an observer, and is effectively hidden from the game developer.

6.2.3 Flexibility / Modifiability

Flexibility and modifiability are important to allow developers to re-use components in a wide variety of games. So by investing money in an expensive piece of domain-specific technology, the developer has not severely limited the kinds of games he/she can make. The proposed architecture is flexible enough to allow developers to mix and match components allowing them to assemble almost any possible game.

This architectural requirement was demonstrated by both the reference games and the prototype. Both very different reference games were able to be designed using the proposed architecture, strongly suggesting the architecture can support a wide variety of games. The prototype also demonstrated flexibility in that attaching different domain-specific components resulted in a variety of “game-like” applications.

6.2.4 Expandability / Maintainability

Expandability and maintainability are important in keeping development time and costs down, and should ultimately result in a better quality upgrade. Iterative game incarnations are most often technology upgrades with minor tweaks in game play, and should not require complete redesigns. The proposed architecture supports this capability

by keeping the domain-specific components independent of each other, allowing technologists to upgrade each system without breaking the other functionality.

The prototype demonstrated this requirement in a few different ways. First, as stated in meeting COTS-based development, the systems are truly independent of each other allowing technologists to modify components without breaking others. Second, the prototype shows expandability by demonstrating technology upgrades by swapping in entirely different systems. The prototype application was able to make the technology upgrade from a 2D graphics system to a 3D graphics system by simply attaching a different component. Lastly expandability is supported by the architecture in much the same way it supports flexibility - systems can be expanded by simply adding a new component.

6.2.5 The Performance Concern

Although Performance was not an official requirement of this architecture because the assumption that most the performance issues reside inside the components, it is definitely something game developers would be concerned about. Reviewing the prototype and the simple design, it appears as though this architecture has very little impact compared to the more monolithic designs presented earlier.

First the architecture allows for direct interface invocation, not requiring any message-handling overhead (although the architecture does not preclude the use of using messages as the method of communication). Second, the architecture doesn't create much in the way of extra communication. For example, whether an object calls a graphics library or the graphics systems requests an object to draw, the number of

interactions is the same. The exception comes from the fact that there is no direct communication between the domain-specific components. When components need to communicate, they must write data to the object management system, and wait for the response in the next system tick cycle.

The prototype seems to support the notion that performance is not significantly affected by the architecture. In quick comparisons between the samples that came with the Ogre™ graphics engine, and the prototype there were no significant performance differences. Although more detailed profiling would be required to prove how much the architecture affects performance; that is beyond the scope of this thesis.

6.3 Important Considerations

Developers considering this architecture should read and understand some of the important considerations that will affect development. These are a few items of wisdom that were found during the work on this thesis.

6.3.1 Design is Critical

One important fact when using this architecture is that the architecture "supports" many of the quality attributes. The design plays a large role in determining whether those quality attributes are part of the system. One such example is the quality of knowledge localization. This quality wasn't realized until the design phase where the observer pattern came into play.

The design can also negate some of the implied quality attributes of the architecture. For example, the architecture also "supports" easy component integration by limiting the communication between the domain-specific component and the object management

component to simply requesting objects, and read/writing data to those objects. The prototype design, where the object component is forced to implement interfaces for each attached component, makes component integration quite tedious. So while it is possible to design complex game systems with the proposed quality attributes using this architecture, it left to the designer to ensure those attributes are realized in the system.

6.3.2 Central Object Management System = VERY different

This architecture uses a very different topology than the designs of today. The current trend seems to be that every system has its own object management system – e.g. graphics & 3D sound engines each have their own way of organizing objects. This makes the libraries easy to use, but it duplicates functionality.

One of the goals of this thesis is to promote COTS based development where specialists can design the best and most optimized components. By centralizing object management into one area, it means specialists can build the best management algorithms, whether BSP trees, Oct trees, etc. without being concerned with some of the domain specialties. It also means people writing the domain-specific components, like sound, need not concern themselves with complex scene management.

Many game developers may take issue with this approach making arguments that items like a graphics engine may have highly optimized scene management specialized for that particular graphics engine, and that a 3rd party scene manager would impact performance. The thing to realize is that this is a design concern, not an architectural concern. The architecture merely states that the object management component will provide a domain-specific component with objects to process. There is no restriction

saying that a specific graphics component can't recommend a specific optimized scene management system to use. By placing it in a central location, however, that same scene management system is available for the other systems to use.

The architecture also doesn't state that there is only one scene management system within the object management component. The object component may have multiple scene managers that the different systems can use. For example one scene manager may be designed to provide a list of objects in the player's view that the graphics engine will use. Another scene manager could exist that is designed to provide a list of objects within a specific radius of the player that is used by the sound and AI components.

6.3.3 Think about the Data

When designing to this architecture it is important to think about the data that will reside in the shared data store. Part of the benefit of this design is that the data you place there is usable by all domain-specific systems. For example, objects in the prototype had location and orientation that was used by both the graphics systems and the AI systems.

Another issue related to data is the data types used. Since the domain-specific components may be written by different companies, and so may be expecting slightly different data types. The graphics engine may want "double" precision floating point values for location, while the sound engine may require integers. While this problem is no different than current games using 3rd party libraries, it shows itself in a slightly different manner.

6.4 Future Research

During the course of completing this thesis a great many ideas were left on the drawing board because they were beyond the scope of this thesis. They are captured here as ideas for future research, and represent many of the interesting problems that remain to be answered.

6.4.1 Can this Architecture Work for Massively Multiplayer Online Games

Massively multiplayer games represent the next big advancement in electronic entertainment. The enormous number of distributed players and objects present some very interesting problems that were not considered in the design of this architecture. It will be interesting to see if this architecture can scale across multiple servers, with thousands of players, all existing in a persistent world.

6.4.2 Design: Domain-specific Component Connection to the Object Management Component

The simple design used for the prototype, forcing the object management component to implement interfaces, is very weak. While forcing objects to implement data access interfaces may be necessary to maintain performance, attaching components and requesting object lists don't have the same restrictions. A better design would allow domain-specific components to easily attach to the object management system, and request objects to process.

6.4.3 Design: No More Interfaces to Access Object Data (If performance allows)

While function calls to retrieve the data is probably the fastest method to access object data this architecture can support, there may exist more generic methods that don't greatly affect performance. For example, if a simple query language methodology could allow domain-specific components to access object data without a significant cost in speed, the ability to add and change components is made significantly easier.

6.4.4 Architecture Inside the Components

While the focus of this thesis was designing the architecture at the inter-component level, architecting the components themselves is still relatively uncharted territory. It would be in interesting assignment to research the domains and see if a common architecture could be created for the specific components. If no such architecture exists, which is likely due to the diversity of the domains, then work should begin designing reference architectures for each of the domains.

6.4.5 What is messaging overhead for independent component style

The independent components and system of system architectural styles were rejected in this thesis because it was thought the messaging overhead were to high for game systems. It would be an interesting experiment to see how much that overhead would affect performance. If messaging does not cause a significant drop in performance many other architectural possibilities are made available.

6.4.6 The Architectural Tradeoff Analysis Method

An important piece of work is left undone in this thesis, and that is the architectural tradeoff analysis method (ATAM). Due to time restrictions not all quality attributes could be analyzed. It would be an extremely worthwhile endeavor to truly analyze this architecture more completely, looking at those quality attributes that were not tested.

Works Cited

"3 Million Lines of Code." EdGames. Sept. 13 2004. <http://edweb.sdsu.edu/courses/edtec670/edgames/2002/12/3-million-lines-of-code.htm>.

"3D Engines Database: Unreal Engine 3." *DevMaster.net*. Sept 14. 2004. <http://www.devmaster.net/engines/engine_details.php?id=25>.

"A $30 Billion Dollar Industry." Aug. 2003. < http://www.xboxcity.com/console/NewsDetail.asp?NewsID=1422&fc=0 >.

"A Brief History of the FPS." Aug. 2003. < http://www.3dactionplanet.com/features/editorials/fpshistory1/>.

Adolph, Steve. "Reuse and Staying in Business." *Gamasutra*. 12 Dec. 1999. Sept 12. 2004. <http://www.gamasutra.com/features/19991213/adolph_02.htm>.

Alves, Carina. João Bosco Pinto Filho and Jaelson Castro. "Analysing the Tradeoffs Among Requirements, Architectures and COTS Components." Centro de Informática, Universidade Federal de Pernambuco Recife, Pernambuco. Sept. 5 2004. <http://www.cs.ucl.ac.uk/staff/C.Alves/WER01_COTS.pdf>.

Bass, Len. Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley, 1998.

Busto, Roberto Del. "Games and Simulations." Aug. 2003. < http://coe.sdsu.edu/eet/Articles/gamessims/index.htm >.

Calvert, David. "Software Architectural Styles." 3 June 1996. Aug. 16 2004. <http://hebb.cis.uoguelph.ca/~dave/27320/new/architec.html>.

"Definition: System of Systems." *The Free Dictionary.com*. Oct. 7 2004. <http://encyclopedia.thefreedictionary.com/System%20of%20systems>.

"Domain-specific Software Architectures." Aug. 2003. < http://sunset.usc.edu/classes/cs578_2003/13-Domain-Specific%20Software%20Architectures%20(DSSA).pdf >.

Duffy, R. "Software Architecture." Sept. 12 2004. <http://members.aol.com/rduffy4187/report.html>.

E. Berard. *Essays in Object-Oriented Software Engineering*. Prentice Hall, 1992.

Fristrom, Jamie. "Manager in a Strange Land: Most Projects Suck." *Gamasutra*. 17 Oct. 2003. Sept. 12 2004. <http://www.gamasutra.com/features/20031017/fristrom_01.shtml>.

"History of Arcade Games." Aug. 2003. < http://www.hut.fi/~eye/videogames/arcade.html>.

"How to Make a COTS Project Fail." Aug. 2003. < http://www.versaterm.com/topic_list/topic16.htm>.

Nilson, Roslyn; Kogut, Paul; & Jackelen, George. *Component Provider's and Tool Developer's Handbook Central Archive for Reusable Defense Software (CARDS)*. STARS Informal Technical Report STARS-VC-B017/001/00. Unisys Corporation, March 1994.

Rollings, Andrew and Dave Morris. *Game Architecture and Design.* The Coriolis Group, 2000.

Sloan, Jason and William Mull. "Doom 3 FAQ." Aug. 2003.

< http://www.newdoom.com/newdoomfaq.php#5>.

“Unreal Tournament History”. Oct. 20 2004.

<http://www.unrealtournament.com/general/history.php>.