Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines?

Eelke Folmer

Game Engineering Research Group
University of Nevada, Reno
89503 Reno, Nevada, USA
research@eelke.com

Abstract. Expanding deadlines and escalating costs have notoriously plagued the game industry. Although the majority of the game development costs are spent on art and animation, significant cost reductions and more importantly reductions in development time can be achieved when developers use off the shelf components rather than develop them from scratch. However, many game developers struggle with component integration and managing the complexity of their architectures. This paper gives an overview of developing games with components, presents a reference architecture that outlines the relevant areas of reuse and signifies some of the problems with developing components unique to the domain of games.

Keywords: Games, COTS, Game architectures.

1 Introduction

Developing games is an expensive and risky activity. Computer games have evolved significantly in scale and complexity since the first game –Pong— was developed in the seventies [1]. Technological advances in console technology, e.g. advances in processor speed, storage media, memory size and graphic cards have facilitated increasingly complex game play and large quantities of realistic graphics. A natural consequence of these advances is that the cost for game development has skyrocketed. Estimates about the average costs for developing a console game range between 3 and 10 million dollar [2]. In addition development time and team size nearly doubled the last decade [3]. An additional problem that developers have to face is the observation that the games is predominantly hits driven; a UK demographics study revealed that the top 99 titles (only 3.3% of development) account for 55% of all sales [3]. The price of computer games, on the other hand, has stayed about the same over the last 10 years and has only slightly increased (from \$50 to \$60) for 3rd generation (Xbox 360 / Playstation 3) games.

As the game industry continues on a path towards longer development times and larger budgets, developers need to find ways to either sell more games or reduce the cost of building games. One way to reduce the cost of games is to reuse particular game components. Rather then reinventing the wheel when developing a 3d engine, a physics engine or a network component, game developers can choose to use an existing Commercial of the Shelf (COTS) Component. The primary motivation for an organization to use COTS is that they will:

- Reduce overall system development costs and development time because the components can be bought of the shelf instead of having to be developed from scratch. Buying the component is usually cheaper as the development costs for the component are being spread out over the multiple game titles in which the component is incorporated.
- A higher quality of components is to be expected as one can assume that these components are being used in different games, in different environments; more rigidly testing and stressing the quality of the component than in a single game setting.
- In addition a COTS based approach benefits the game industry as a whole as successful COTS developers can focus on one particular aspect of a game e.g. physics or 3d engines. This allows them to advance this technology at a faster rate than when they were building games. These advances are then available for more games to use [1], creating a win-win situation for everybody.

COTS development is not new trend in the games industry. In the past a significant number of games have been built upon existing technologies. Especially in the first person shooter (FPS) genre tech is heavily being reused. FPS engines like the Doom™ engine by ID games and the Unreal™ engine by epic games have spawned numerous successful games. However COTS have predominantly focused on the 3d rendering engine technology and or well understood sub domains such as audio and networking. Ten years ago only a handful of commercial game engines were available and only a small number of libraries for audio and networking. Because of the rapid evolution of games the last decade, game developers now can choose from a plethora of components dealing with various aspects of games e.g. rendering, object management, physics, artificial intelligence and so on. Being able to choose from a multitude of components (some of which are open source and hence free) is good for the game development community as it will allow significant cost reductions and time to market and will allow game developers to concentrate on the features of their game rather than on generic features common to all games. However the success of component based development as can be concluded from other domains such as the web domain, will largely depend on how easy game developer can incorporate such components in their games. In this paper we explore COTS based game development and identify some of the issues that developers face when adopting COTS based game development. The remainder of this paper is organized as follows. In the next section, we present a reference architecture that allows us to identify relevant areas for reuse. Section 3 discusses the relevant areas of reuse. Section 4 discusses some of the problems that hamper component based game development and discusses some research questions worth investigating. Section 5 concludes this paper.

2 A Reference Architecture for Games

Before we discuss the different components available to game developers we need to provide a common vocabulary with which to discuss different game implementations and commonalities between those game implementations. In order to understand which parts of a game are specific and which are general we propose a reference model that allows us to understand the separations and relations between the different parts of a game design. The highest level abstraction of any software system is called the software architecture i.e. the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution [4]. The software architecture is an important artifact in the development of any system as it allows early analysis of the provided quality of a system such as performance, maintainability. This activity is important as these qualities are to a certain extent restricted by its architecture design and during architecture design one can still cost effectively change design decisions. As a specific domain of software systems ages and matures, more and more systems are developed from different organizations, and their functionality, structure, and behavior become common knowledge e.g. abstractions or software architectures will surface that represent their common denominator [5]. Such an abstraction is called reference architecture, which in essence is a software architecture, at a higher level of abstraction. A reference architecture does not contain any implementation details so it can be used as a template solution for designing new systems. Another benefit of having a reference architecture is that it can point out potential areas for reuse.

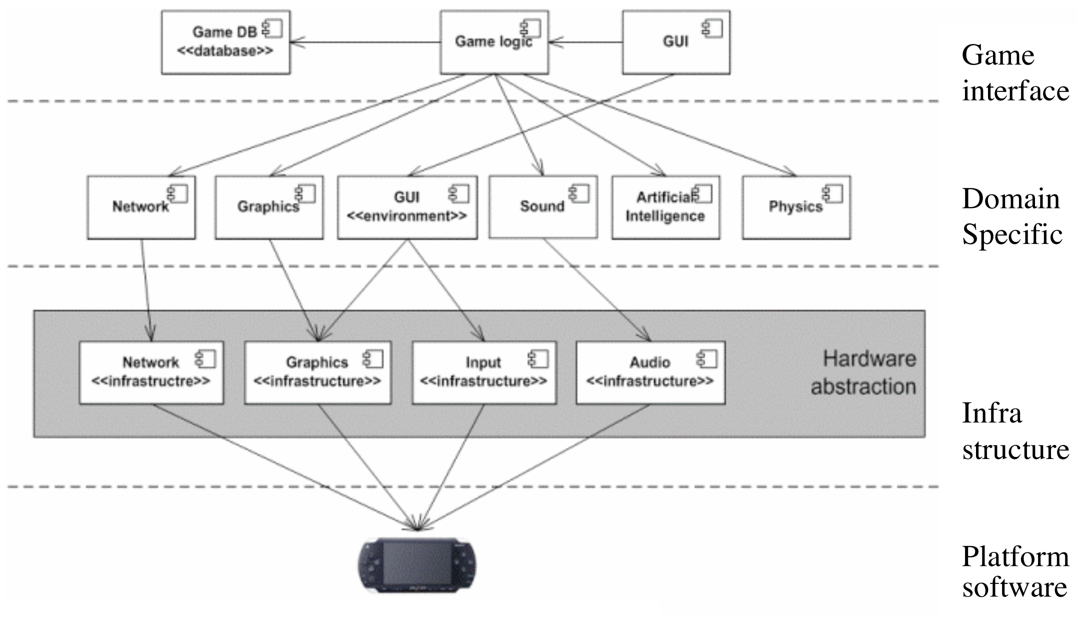

Fig. 1. A reference architecture for the games domain

We derived a reference architecture (RA) from two published game architectures [1, 6], an RTS system which has been published [7] of which we extracted an architecture design and a number of unpublished/ undisclosed systems. Our reference architecture is inspired by the layered reference architecture for component-based development as proposed in [8]. Their layered reference model consists of five layers; the interface, application, domain, infrastructure and platform layer and it puts the most specific components in the highest layer and the more general reusable components in the lower layers. To create our reference architecture we looked at different game architectures, we analyzed their components, and we then analyzed the commonality of these components across different game architecture implementations -and different game genres. Finally these components were organized according to the layered architecture reference model proposed in [8]. We left out the application specific layer form their model. This has resulted in the reference architecture displayed in Figure 1. Our reference architecture consists of four layers:

- *Game interface layer*: the top layer in our reference architecture is comprised of objects and components, which encapsulate the game logic. In this layer all the game specific objects are found such as models and textures. The game user interface, the game logic and a set of specific game objects (models, textures) usually stored in a file system or database. The objects in the database are part of this layer but the database functionality is provided by components from the infrastructure layer. For reasons of simplicity we didn't make this connection explicit.
- *Domain specific layer*: This layer is comprised of components, which encapsulate the interface to one, or more classes, which are specific to the domain of games. Examples of such components are usually graphics, physics, network, sound etc. These components are generally used from multiple places within the game. Behavior of game objects such as determined by the AI or physics is usually controlled by scripting languages such as lua or python that are part of the infrastructure layer.
- *Infrastructure layer:* This layer is made up of bespoke components that are potentially re-usable across any domain, providing general-purpose services such as input/output, persistence, database management, scripting communication, hardware abstraction etc.
- *Platform software*: this is comprised of standard or commonplace pieces of software that are brought in to underpin the game.

The validity, accuracy and completeness of this RA are open for discussion. Our RA has only been based on a limited number of available game architectures, which might not represent an accurate cross section of all possible game architectures. The architectures we derived this from did fit in this RA. Game companies tend not to disclose the architectures of their games. Usually a RA also defines stakeholders, different views and supported qualities and usually the RA is analyzed for its support of those qualities. In this paper we merely outline the RA to sketch out commonalities between different game architecture implementations and point out potential areas for reuse.

3 Areas of Reuse

As can be seen in our reference architecture six areas of reuse can be found in the domain specific layer:

- **Network -** Focuses on the communication between games and servers.
- **Graphics –** A collection of subsystems all related to visualizing the game.
    - **Rendering** - Provides basic 2 or 3 dimensional rendering (producing pixels) functionality.
    - **Modeling -** Focuses on abstract representations of game objects and managing those objects e.g. scene graphs.
    - **Animation:** functionality related to creating moving images.
    - **Texturing& effects:** functionality related to applying textures and light effects to particular models.
- **GUI –** Provides the functionality to build game interfaces.
- **Artificial intelligence -** Provides functionality related to produce the illusion of intelligence in the behavior of non-player characters (NPCs), such as path finding.
- **Physics -** Provides physics related functionality such as collision detecting e.g. game objects should adhere to Newton's laws of dynamics.
- **Sound –** libraries for modifying / generating sounds playing mp3's etc.

Usually a game engine provides a number of such components combined in one, however game engines are usually designed for a particular game and might not be suitable for what your game needs. Numerous third party components can be found which provide a plethora of functionality. We don't provide an overview in this paper but a complete overview can be found on http://www.gamemiddleware.org. To provide a complete picture another important area of reuse should be mentioned that are not included in the reference architecture.

- **Tools –** Tools (such as exporters and importers between different graphic applications) are not part of the game itself but are reused between games. The tools side of game development is unique and important .The tools may require twice the amount of code and are a huge detail given the number of content producers teams have these days. Usually numerous content generation tools such as 3D studio Max or Maja are used but developers often end up having to write numerous plugins and converters to be able to port models/ graphics from such tools to their game engines, which is quite cumbersome.

4 Problems with COTS Development

We identified the following problems possibly limiting the success of COTS.

4.1 Components Versus Frameworks

The success of component based development in the domain of games will depend on how easy developers will be able to integrate existing components into their games. Looking at other domains such as web-based systems, COTS were never as successful as they were claimed to be. COTS were considered to be the "silver bullet" [9] of software engineering during the nineties but the development with components came with many not so obvious trade-offs; Overall cost and development time were reduced, but often at the expense of an increase in software component integration work and a dependency on a third-party component vendors. As a result, COTS were gradually absorbed into higher granularity building blocks, i.e. application frameworks such as .NET or J2EE which don't come with integration problems but also do not offer much flexibility in the choice of components. A similar argumentation holds for the game industry; game engines for FPS were among the first reusable components. As the game industry matured more and more highly specialized components became available for specific sub areas such as physics and artificial intelligence. We are at a point now that if you want to build a game from components a large number of components need to be integrated --which is not an easy task. There seems to be a movement in the game industry towards developing frameworks. The obvious tradeoffs that need to be made here is that building from smaller pieces gives more control but using a large framework usually gives you the tools and less hassle with integration. More research needs to be done to provide developers with guidelines on how to successfully integrate components.

4.2 Complexity and Architecture Design

Another complicating factor is that games have increased in complexity, a 3d engine 10 years ago was an order of magnitude simpler to understand than it is nowadays. One reason for this complexity might be because more and more components are used. Since COTS developers try to design their component in such a way that it might provide a best overall fit for a large number of games, it means that thick glue layers may be needed to make up for the poor fit that the COTS provides for your game. An example of a glue code is for example the code required to perform data conversions between game components such as rendering or physics who require data to be in a specific format [10]. Glue layers usually become a bottleneck when performance is critical, as lots of data needs to be converted runtime. In addition game architectures are overly complex and do not provide maintainability and flexibility because of the spaghetti of dependencies that exist between COTS [1]. Components such as a renderer, physics, audio and artificial intelligence all need their own local data management model (with varying degrees of detail) such as binary spatial tree where the state of game objects is stored. When the state of a game object changes in any of the models this needs to be updated in all the associated models, leading to a synchronization and overhead between components. Another complicating factor is the object centric view that most games adopt [5]; Games are composed of game objects such as entities like cars, bullets, people representing real life objects. Game objects are responsible for all their own data manipulation and most COTS are just functional libraries that help the object do what its supposed to do. With the increase in complexity of this functionality the COTS objects become large and complex and unwieldy [1]. Current game architectures do not support COTS development very well and possible alternatives such as data driven or black board game architecture as proposed in [1] need to be further investigated with regard to performance, scalability and the desired maintainability and flexibility for component based game development.

4.3 The “emerging” Architecture

Usually game developers pick a game engine and write the necessary glue code to incorporate the desired COTS. If we develop our game like this a software architecture "emerges" rather than is designed upfront. An architecture consists of components and connectors and usually some design rationale. An architecture is mainly used as a tool to communicate design decisions to software engineers and it highlights the system's conceptual properties using high level abstractions which allows early analysis of quality requirements. In this model COTS can be used as solutions which facilitate such a design. The danger with randomly assembling a game using components is that the resulting architecture might not be the most optimal given the games quality requirements. There are still some degrees of freedom with regard to component composition that are often unexplored. Software Connectors play a fundamental role in determining system qualities; e.g. the choice to use shared variables, messages, buffers, calls or table entries has a big effect on the qualities of the game such as performance, resources utilization and reliability. Abstracting and encapsulating interaction details may help fulfill properties such as scalability, flexibility and maintainability, which may help reduce the complexity of game architecture designs. With regard to game design this area needs to be further explored.

4.4 The Buy or Build Decision

Because incorporating COTS is difficult and game architecture are complex, deciding which component to select to use in your game is a difficult decision. Especially for game related components usually deep technical knowledge is required to understand how to successfully use and integrate the COTS [10]. Game development requirements are very volatile and change frequently as a result some game developers end up rewriting most of the functionality that they need from the component and they would have been better of building the component themselves in the end. In order for a COTS to be successful it needs to be designed in such a way that it facilitates many needs, so it can be used in many different games. But as it is often impossible to fulfill everyone’s needs the COTS need to provide a most common denominator of the required functionality that might not be the best fit for what your game needs. It will take some time to understand the component yet there is no guarantee the COTS will actually speed up the development if after a long investigation the COTS proves to be a poor fit and so much functionality needs to be rewritten that it was better to develop such a component from scratch and avoiding things like ad hoc programming and design erosion. Guidelines for analyzing components and strict interface agreements might mitigate some of this risk but need to be further explored.

5 Conclusions and Future Research

Developing games with components has the potential to minimize development costs and speed up development time. However, currently game developers struggle with a number of problems such as how to successfully integrate the component in their game. Deciding whether the component provides what is required for the game. Managing the complexity of their game architectures and analyzing whether the architecture that results from component composition meets the required quality requirements. Our future research will take a closer look at component composition by doing a comparative study on the relative ease of integration for a number of open source components for a Real time strategy game engine for AI research that is currently being designed at the University of Nevada. These experiences will allow us to develop a set of guidelines and or a game architecture that might facilitate developing games with components.

References

1. Plummer, J.: A Flexible and Expandable Architecture for Electronic Games. Vol. Master Thesis. Arizona State University, Phoenix (2004)
2. Grossman, A.: Postmortems from Game Developer. CMPBooks, San Francisco (2003)
3. DTI: From exuberant youth to sustainable maturity: competitive analysis of the UK games software sector. (2002)
4. IEEE Architecture Working Group. Recommended practice for architectural description IEEE (1998)
5. Avgeriou, P.: Describing, Instantiating and Evaluating a Reference Architecture: A Case Study. Enterprise Architect Journal, Fawcette Technical Publications (2003)
6. Andrew Rollings, D.M.: Game Architecture and Design. Coriolis Technology Press, Arizona (2000)
7. Michael Buro, T.F.: On the Development of a Free RTS Game Engine. GameOn'NA Conference, Montreal (2005)
8. Mark Collins-Cope, H.M.: A reference architecture for component based development
9. Brooks, F.: The Mythical Man-Month; Essays on Software Engineering; Twentieth Anniversary Edition. Addison-Wesley, Reading (1995)
10. Blow, J.: Game Development: Harder than you think. ACM Queue (2004)