Modular Architecture for Computer Game Design

Teddy McNeill
Thomas Jefferson High School for Science and Technology
Alexandria, Virginia

June 11, 2007

Abstract

Common current game architectures limit program flexibility and modularity. With the advent of middleware and the increasing complexity of games, this is a significant hindrance to development. In this project I have attempted to design and implement (using C++ and OpenGL) a highly modular, data-centered architecture based on the "System of Systems" approach. The final implementation was not required to have any significant complexity within each system (e.g. graphics, AI, etc.) but rather had to demonstrate the successful interaction of independent systems.

1 Introduction

Current game architectures are designed with Object-Oriented programming in mind. This often involves giving the game entity objects the functionality to do all the drawing, calculation, etc. involved in their use. While this method conforms to Object-Oriented programming practices, it has major drawbacks. Specifically, this technique limits the recycling of code and implementation of middleware (or the Component-Off-The-Shelf [COTS] approach). For example, if a sequel to a game were to be created that switched from 2D to 3D graphics, it should not be necessary to completely remake the game; however, that is what is often required.

This project attempted to implement an architecture for games that allows efficient reuse of code and accommodates the COTS approach. This primarily was based on the separation of data and calculations. The architecture used a data-centered, System-of-Systems organizational structure.

The widespread use of this architecture would allow game developers to make extensive use of middleware during the creation of almost any game. It would also reduce the time and work required to produce games with similar elements. While the direct application of this research is to games, the architecture and COTS approach can be applied to nearly any form of software development; the reuse of code and friendliness towards middleware could expedite the development of all programs to some degree.

2 Background

Jeff Plummer, in his paper A Flexible and Expandable Architecture for Computer Games, provided the inspiration for this research. Plummer attempted to solve the problem of game developers having to rewrite large sections of game code that are designed to create very similar outputs. The proposed solution was the System-of-Systems approach.

The System-of-Systems approach is data-centered, meaning that all classes are passed a pointer to a class containing all the data that represents the game world and the entities within it. All classes exist to operate on this data. This is as opposed to more common systems that would allow objects (entities) to operate on their own data.

The System-of-Systems approach, as its name indicates, considers a game no more than a group of interacting systems. Each of these systems (such as AI, Graphics, Collision Detection, Game Logic) can be represented as a class. Each of these classes would act upon the central data.

What this approach provides for is the total separation and independence of the systems. That is, within a correctly designed architecture, the Physics class need not know what it is acting on to operate properly. Thus, with the correct interfacing of classes, a middleware physics system could be added with little effort. The same could be done with nearly any system excepting Game Logic.

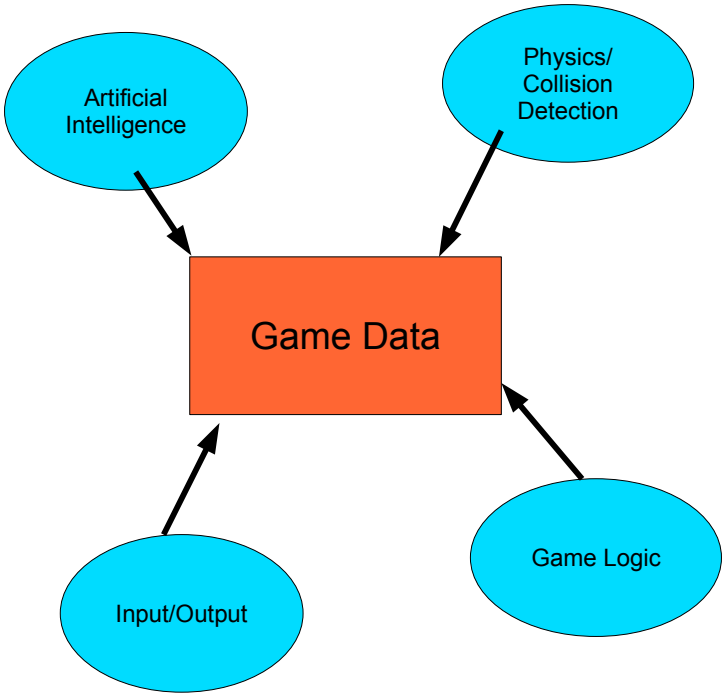

Figure 1: The System-of-Systems architecture as implemented in this project.

3 Development

3.1 Requirements

The requirements for the project implementation were fairly skeletal. It was necessary to include several systems in order to examine their interaction and interfacing and in order to provide proof of concept. However, it was not necessary that any of these systems exhibit any complexity within themselves. For instance, the graphics system could only display cubes and the architecture could still be usefully examined.

The only other requirement was that the creation actually constitute a 3D game (rather than some other non-game type of program) in order to ensure relevance to the problem being addressed.

3.2 Development Plan

The benefits of the System-of-Systems architecture are focused in the development process rather than in the final product of the code. Therefore, in order to evaluate the effectiveness of the architecture, it was necessary to

add a new system into a fully-functional game. This would be analogous to adding in a piece of middleware. Therefore, the implementation of the project came in three major stages. First, I prototyped the technologies necessary for the game. Second, I restructured this code into a fully functional System-of-Systems game. Third, I added a brand new system into the existing game and analyzed the ease of this addition.

3.3 History

3.3.1 Prototyping

Due to the author's lack of experience with the tools at hand (C++ and OpenGL), the first period of development was spent on prototyping the necessary functionality for a game. These mainly related to specific OpenGL functions and basic programming for a first-person 3D game environment. By the end of this first stage of development, a basic 3D first-person target-shooting game was implemented.

3.3.2 Structuring

The second period of development began with organizing and cleaning up the code that had already been written and designing the specifics of the new architecture. This design process experienced early difficulty due to the limitations of the tools being used. Specifically, the use of the GL Utility Toolkit (GLUT) required that the input, output, and initiation of the program occur in the same class. After this was taken into account, the design was completed. It was decided that the final architecture would make use of four classes: a main Input/Output class, a Data class, a Physics/Collision Detection class, and a Game Logic class. The I/O class contained the GLUT functions, the Data class contained the data that described the game world, the Physics class governed the movement of game entities, and the Game Logic class dealt with game-specific rules such as scoring the player's performance.

3.3.3 AI Addition

After the new design was implemented, it was necessary to add a new system in order to examine the architecture's usefulness. I chose to add an Artificial Intelligence system, along with new Enemy data. This effectively

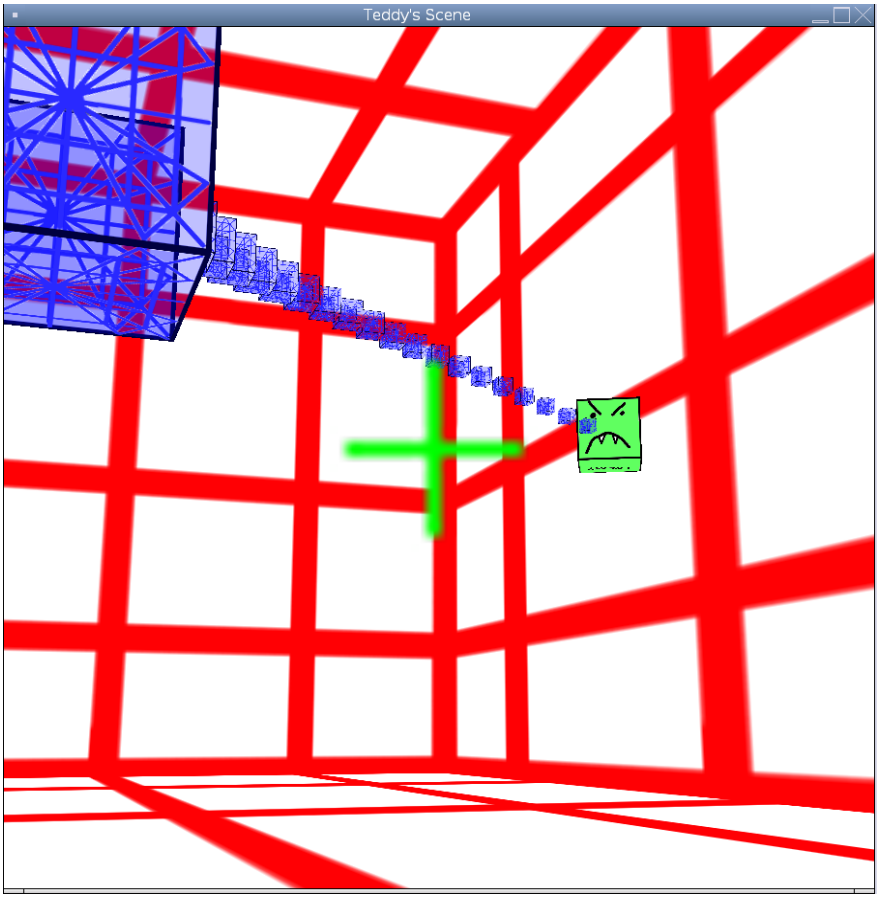

Figure 2: A screenshot from the completed game. Note the Enemy entity and its shots. This is being governed by the new Artificial Intelligence class.

changed the game from a target-shooting game to a more standard first-person shooter. This stage of development required that I alter the I/O class in order to display the new Enemy entity. I also had to add new Data, change the Game Logic in order to accommodate the new scoring system, and add Physics functionality for the new interaction with the Enemy entity and the projectiles it would fire at the player.

4 Results and Discussion

4.1 Overview

The purpose of this project was to implement a modular architecture and examine its development and benefits. The completion of such an architec-

ture could provide a base for future development of complex games requiring middleware or reuse of code. The implementation required that several independent systems operate on a piece of central data, but each of these systems could be very primitive. While the final program of this project was not entirely successful at demonstrating a working System-of-Systems architecture, the project was successful at further clarifying what is necessary to make such an architecture work effectively.

4.2 Results

The key to understanding the effectiveness of this program's architecture lies in the addition of the Artificial Intelligence system that came at the end of the development process. In a proper System-of-Systems architecture, one would be able to write the new system and integrate it into the rest of the program with minimal modifications made to the other systems. This was, unfortunately, not the case with the AI addition.

The problem with my implementation was that there was not enough abstraction. That is, each class would operate on individual variables or lists of variables (such as the player's position, or the projectiles that the player generated). Therefore, when the Enemy was added, the I/O class required extra code to display it, the Physics class required new functions governing its collision detection, et cetera. This limitation meant that the addition of a new system was not seamless or easy for the developer, but rather required nearly as much modifications as another architecture would need.

4.3 Analysis

The solution to the problems that this project discovered is to abstract all operations upon the game data. For example, rather than display individually the shots, the world, the targets, et cetera, the I/O class ought to run through a list of "displayable" items and draw them in a more automatic fashion. With such a system, the inclusion of a new entity (such as the enemy) could be facilitated simply by adding this new entity to a list in the Data class, with minimal or no modification necessary to the I/O class.

One effect of this solution is that the newly included classes would require greater standardization. If the I/O class is to draw a new entity or the Physics class is to know how to move it in the game world, the new entity must keep information that pertains to these functions, and it must keep this

information in a form that the existing classes "know" how to access. This is achieved by storing such variables in a standardized manner that is known to the developers of all systems and data.

It should be noted that despite the troubles with interfacing new data with the existing classes, the data-centered nature of the architecture was of great help during development. All of the systems, including the Artificial Intelligence system, were implemented more conveniently and efficiently due to this structure. A wide range of variables was easily accessible to all systems. Also, the basic separation of systems was both simple and effective; different systems were independently designed and implemented, and this allowed easier implementation of all.

4.4 Conclusion

The System-of-Systems architecture still has great potential. Though this project was not fully successful in implementing a working instance of the architecture, the benefits that were attained through even the more limited implementation show that this structure would accomplish its goals of supplying modularity and flexibility. This researcher suggests a follow-up project attempting to fix the faults and troubles encountered in this implementation; such research, especially if followed by projects that demonstrate major game modifications, could conclusively demonstrate the superiority of the System-of-Systems architecture.

References

[1] Plummer, Jeff, "A Flexible and Expandible Architecture for Computer Games", December 2004. http://www.gamasutra.com/education/theses/20051018/plummer_thesis.pdf (September 19, 2006)

[2] Amato, John, "Collision Detection", GameDev.net September 15, 1999. http://www.gamedev.net/reference/articles/article735.asp (January 16, 2007)

[3] Kershner, Jeff, "Object-Oriented Scene Management", GameDev.net May 2, 2002.

http://www.gamedev.net/reference/articles/article1812.asp (January 16, 2007)

[4] Nicollet, Victor, "Item Management Systems", GameDev.net October 21, 2004.
http://www.gamedev.net/reference/articles/article2163.asp (January 16, 2007)

5 Appendix

5.1 Main Loop Code

This is the point within the main loop where all the calculations of the game world take place. Note the simplicity of this step due to the System-of-Systems architecture.

```
physics.tick(&d);
ai.tick(&d);
game.tick(&d);
```

5.2 Physics System Code

This code demonstrates the manner in which the data-centered structure was used.

```
for(int i =0;i<150;i++)
{
    if(d->shots[i].exists && distance(d->shots[i].pos,d->enemy.pos)<=1.0)
    {
        d->shots[i].exists=false;
        d->currentcollisions++;
    }
}
```

5.3 Graphical Output Code

This code demonstrates the lack of abstraction that ultimately crippled the System-of-Systems benefits. Note how specific drawing routines are used for the shots of both the player and the Enemy entity.

```
glBindTexture(GL_TEXTURE_2D, textures[2].texID);

for(int i =0;i<150;i++)
{
    if(d.shots[i].exists)
    {
        Vector3 pos = d.shots[i].pos;
        drawCube(0.2, pos);
    }
}

for(int i =0;i<150;i++)
{
    if(d.enemyshots[i].exists)
    {
        Vector3 pos = d.enemyshots[i].pos;
        drawCube(0.2, pos);
    }
}
```