University of Nevada, Reno

Towards Automatic Parallel Game Engine Architectures

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in
Computer Science

by

David A Carr

Dr. Eelke Folmer/Thesis Advisor

May, 2009

THE GRADUATE SCHOOL

We recommend that the thesis

prepared under our supervision by

DAVID A CARR

entitled

Towards Automatic Parallel Game Engine Architectures

be accepted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

Eelke Folmer, Ph.D., Advisor

Frederick C. Harris Jr., Ph.D., Committee Member

Jonathan Weinstein, Ph.D., Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

May, 2009

Abstract

As video games steadily increase in complexity and detail, game engines must also improve to be able to support both rapid and modular development while still maintaining high performance. Component-based architectures have been shown to be effective for allowing developers to rapidly create stunning games with modular components, but unfortunately come with a performance cost over traditional call-and-return systems. This thesis proposes a method that modifies existing component-based game engine architectures to automatically distribute and synchronize work in order to improve performance.

Contents

List of Figures

1 Introduction

Every year video games become more complex and more demanding on hardware. Massive efforts are made by game development companies to maximize performance with limited resources. Game companies strive to minimize their effort in developing each game by either developing or buying game engines that contain many of the reusable parts of a game.

When developing these game engines, it is important to maximize both performance and flexibility. Performance is important because it will allow the complexity of games to increase without affecting user experience. Flexibility is important because the more times that a game engine can be used, the more cost effective it will be. If a game engine can be used for a long time and with many different types of games, it will represent a smaller cost for each game that it is used for.

Unfortunately, the two needs of performance and flexibility are often opposed to one another. In order to maximize performance, it is often necessary to make assumptions about the type of game being developed and the platform being developed for. These assumptions limit the applicability of the game engine and thus reduce its effectiveness. On the other hand, when maximizing flexibility it becomes difficult to apply the specialized algorithms and structures that have been developed for many years for games [10]. There is no accepted best practice for solving this interdependence of performance and flexibility, and it is an area that calls for further research [14].

One method that is currently being explored to balance the needs of performance and flexibility is component based architectures [9, 11]. In these architectures, each entity in a game world is composed of many pluggable components, and all components interact with a central system instead of directly with each other. With this kind of decoupling between the different

components of a game, it is possible to provide default components to accomplish very common tasks, but it is also possible to change out components to accomplish more specialized tasks [11].

With the flexibility of these component-based engines, however, there comes the cost of the overhead of having all components talk to each other through an intermediate system layer [11]. In this thesis we propose a method by which a component based game engine can be modified to automatically run the different components in parallel while still keeping all of their interactions synchronized. There are two implementations of this method covered.

The first implementation simply takes the simulation and splits it up into several independent simulations, with each simulation processing its own subset of the overall task set and synchronizing its state with that of its peers. This method has a thick management layer that does all synchronization and distribution across the network and keeps all nodes working together. Unfortunately, the performance of this system was not good because of the complexity of this synchronization and because of the lack of explicit data flow between the different processing units on different nodes.

The second implementation addresses the shortcomings of the first by minimizing the impact of the communication layer on performance while still maintaining the independence of the different components in the system. This implementation also addresses the issue of synchronization encountered between the different nodes of the first implementation by creating an explicit data flow that can be controlled and optimized within the central system. This system has only been developed to a proof-of-concept level so far, but has shown great promise and is proven to perform the basic tasks of task distribution and data synchronization much better than the first synchronization and also better than a simple procedural game engine.

By looking at the state of the game industry, finding a method that could be used and improved, and iteratively approaching an effective solution, we have made significant steps towards the development of an automatically parallel game engine. With the ever-increasing

trend towards massively parallel computation both in home computers and game consoles, this is the area that will have to be exploited soon in order to keep game and simulation performance moving forward. This engine represents an iterative step towards the goal of a fully parallel game engine.

The rest of this thesis will examine the development of two parallel game engines. Each are based on the component-based game engine architecture and expand on that architecture to run in parallel. The first implementation, SDSE, is a modification of the EVA system developed our Chris Miles and runs in networked parallel. The second implementation, SimQ, is a multithreaded application with a game-oriented task queuing system.

2 Background

Video games have become a huge part of modern culture. Each year games make millions of dollars and influence millions of people. In addition to being a huge market, video games are also a rapidly changing one. Games must be innovative in order to be competitive, and in order to be competitive they must be rapidly developed in order to beat competitors to market. Many years ago, games were small programs developed by one or sometimes two people. Recently, however, games have become enormous multi-million dollar projects requiring hundreds of people and years of work. Any effort to reduce this huge demand for resources is bound to reduce cost and improve the productivity of developers.

To this end, game engines have been developed. In short, a game engine is middleware that exists to simplify game development by taking care of the mundane tasks required for most games. In practice, this definition can mean almost anything. Some will tell you that a game engine is a software layer that abstracts away graphics rendering code, while others will tell you that it handles basic scene entity management. At the other end of the spectrum, large commercial game engines seem to be able to handle anything and everything, including graphics, physics, AI, scripting, sound, networking, scene management, animation, and a host of other tasks. From a software engineering point of view, the flexibility of a general solution to the problem of developing a game engine is one of the factors that makes it most interesting.

In addition to providing a wider area of applicability, flexibility also has the advantage of providing maintainability in a software project. This maintainability is provided by the decoupling of the different components of the system and the ease with which a developer can replace or redesign any component [15, 18]. Since computer games are constantly evolving and

becoming more complex [16], this is a very important consideration when developing a computer game.

Consider the example of a game being developed over a long period with many complex components. The graphics part of this engine is finished early on, but it takes longer than expected to develop the artificial intelligence parts of the game and delivery is pushed back. Due to the constantly advancing nature of computer graphics, it is necessary to improve the visual display of the game before shipping. If the graphics code is deeply rooted, this change can involve even further delays and huge development cost. On the other hand, if the graphics code is modular and doesn't affect other components of the game, it can be easily redesigned without having to rebuild other components.

The other key factor that must be considered when developing a game engine is performance. At their very core, video games are interactive real-time simulations of one quality or another. This means that at the end of the day what a game does is take input from the user, pass it through the model of the interactive game-world, and report the result back to the user. This core of immediate interactivity drives the need for performance in games. Indeed, many of the primary advances in game technology haven't been for things such as novel input methods or engaging gameplay, but instead have focused on squeezing every last drop of blood from overstressed processors. This has led to a plethora of specialized algorithms and techniques for everything one needs to make a game, including graphics rendering, scene management, AI, and a variety of other time or memory-intense problems.

The twin challenges of flexibility and performance are directly opposed to one another. In order to get a high amount of one, it is necessary to sacrifice part of the other. To this end, many different game engine architectures have been explored in the past and many will be explored in the future. By examining what challenges have arisen in the past and what solutions have been

proposed for those challenges, it is possible to form a coherent idea of a comprehensive solution to the problem of game engine design.

It is true that Console and PC system architectures are also steadily increasing to meet performance needs. But the direction of the development of this hardware is towards parallel-processing systems such as the one discussed in [12].

2.1 Motivation

In the thesis, "Power and Peril of Teaching Game Programming" [6], two examples of teaching game programming in a classroom environment are given. In the first example, students are presented with the idea of learning about game design during the first half of a semester and implementing a game in the second half. Unfortunately, this approach does not go well. This is for various reasons, but the primary reason is that the students were presented with the abstract concepts of game design instead of being taught practical skills, and then were faced with developing an actual game. With no practical skills or knowledge about how to go about this project, the games that they developed were far from complete. On the other hand, these games were constructed around solid, playable ideas and had great potential.

In the second example of this paper, the authors describe an intensive summer program during which students received a crash-course in the practical aspects of game development and developed a game. This course produced more polished projects than the first, but the creative drive from the first class was unfortunately missing. It is clear from these two examples that it is very difficult to teach the creative and practical skills necessary to develop a complete game to students within one semester. In order to prevent these issues from arising, it would be ideal to minimize the technical knowledge necessary to construct a game so that students could learn sufficiently the creative skills they needed and then quickly pick up the technical abilities necessary to bring those dreams to reality.

A game engine, and specifically a flexible and learnable game engine, is an ideal solution to this problem. With an intuitive data-driven framework, as will be discussed later, many of the technical difficulties of implementing a game can be hidden from the user and automated to a maximum extent.

The paper "Game Engines in Scientific Research" [7] shows how commercial game engines are transforming the world of scientific research and simulation. In the past, scientific institutions paid large sums of money in order to have the latest and greatest in computational power. While it is still true that in order to do very high-fidelity simulations it is still necessary to have a state-of-the-art visualization system and a supercomputer in the back room, many scientific research projects can be improved through the use of existing game engines. These engines offer several advantages over ground-up design of software. The most important of these advantages is the reduction of implementation costs. By implementing a research simulation on the existing platform of a game engine a project can save many months of effort [7].

Another benefit gained from the use of a game engine in the implementation of a scientific simulation is that of performance. Game engines have been evolved over many years to have the highest performance possible, and the algorithms and techniques used to accomplish this are non-trivially complex. By using a game engine it is possible for a researcher to have his or her simulation run much faster than if it were developed in-house, and that reduction in run time can improve the quality of research results and allow greater exploration of the topic being researched.

In the thesis "Multi-threaded Rendering and Physics Simulation" [8], a method useful for decoupling and multithreading the processing of physics and graphics for a simple simulation is shown. Even in the very simple simulation of spheres bouncing in a box shown in this thesis, the development required to accomplish a multithreaded implementation is considerable. In order to have the simulation and rendering be independent, the author is forced to use many locks and

other cumbersome multithreading techniques. To the average programmer these techniques are counter-intuitive, and to the average game creator they are completely indecipherable.

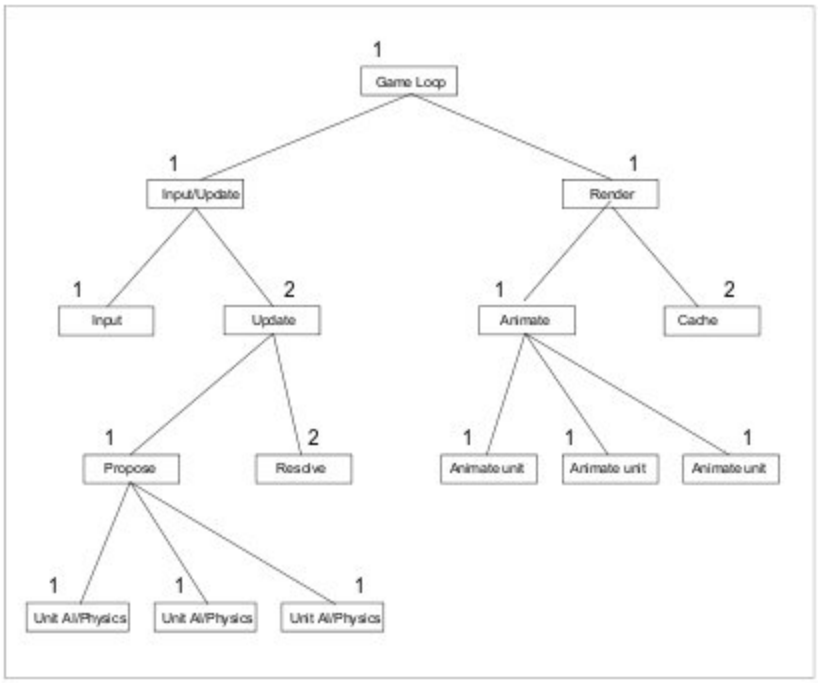
This is a serious problem and highlights the tradeoff between flexibility and performance when designing a game engine. In order to have higher performance, it is useful to use a parallel solution in order to take advantage of today's common multi-core computers. On the other hand, the current technology available for accomplishing multithreading is difficult to learn and use, making it inaccessible to any but specialized programmers. An ideal solution to the problem would be a game engine that allowed developers to declare the data and interactions of their game and used that data to automatically create a parallel implementation of the game. This would maximize both performance and learnability.

2.2 Method

It is shown in [13] that the general processing steps of a game can be broken down and expressed as a cyclic dependency graph. Furthermore, this thesis shows that this graph can be theoretically parallelized with a performance improvement.

In order to address the needs of game engines, it is clear that performance and flexibility must be maximized. There is a known tradeoff between these, but with a clever implementation it should be possible to minimize the compromises made between the two primary concerns of game engine development. In order to accomplish high performance, it is necessary to implement a parallel architecture that will work well on today's multi-core machines.

This kind of parallelization takes two primary forms: task parallelism and data parallelism. In task parallelism, a large group of tasks is broken up as much as possible, and within the various dependencies between those tasks the tasks are executed in parallel. In data parallelism, on the other hand, a simulation is broken up into many pieces of data that are acted upon by processes, and those processes are executed in parallel within the interdependencies of the data. Both of these approaches have their difficulties, and both call for unique solutions.



2.3 Task Parallelism

In order to accomplish task parallelism, it is necessary to accomplish three things. First, the overall task at hand must be broken up into as many atomic tasks as possible. Next, the dependencies between those tasks must be defined. This means that if one task A requires the output of task B to execute, then task B must execute before task A. Finally, a pool of workers will take tasks as they become available for immediate execution and run them, filing the results. This is a common method for doing parallel processing, and has been adapted by several authors to game engine design.

The paper "Multi-threaded Game Engine Design" [4] describes a method for adapting the tasks performed by a game engine to a multi-threaded environment. In order to do this, a system is used where the various tasks are enumerated into a "task tree", as shown in Figure 2-1.

Figure 2-1 An example of task parallelism. [4]

In this tree, all tasks are broken up into sub-tasks until non-ambiguous atomic units are reached at the leaf nodes. These tasks include many things such as gathering input, updating game-world objects, and rendering graphics. By splitting all actions up into a tree, actions are

implicitly grouped by dependency. This means that all update actions must take place before render actions and so on. To process this tree, a thread pool is created that holds a list of worker threads that take actions from the currently processing group of leaf nodes and process those nodes before moving on to the next group of leaf nodes. This method works well for parallelizing tasks, and is shown by the author to be adaptable to the application of game engines.

The thesis "A Game Engine on Distributed Systems using CORBA" describes a method by which the Awe component of a chess program can be parallelized. CORBA is an accepted parallel processing architecture where objects are represented “location agnostic”, so that they can be accessed from anywhere on the network transparently [20]. This approach to the problem of chess Awe is classical and uses the negamax algorithm to search the possible space of board states. The novel part of his solution, however, is the use of parallel processing to speed up the exploration of the game tree. In order to do this, the author breaks the processing of the state tree into discrete tasks and processes them with the use of the CORBA parallel processing system. This application, while showing that task parallelism is applicable to game engine design, is limited in its implementation to a specifically chess game engine [5]. Please note that there are many different implementations of task parallelism, but that this is a fairly

2.4 Data Parallelism

The idea of parallelism is that some tasks are composed of the same operation being performed to many different parts of a large group of data. This idea is very applicable to game engines since game engines operate on spatial, partitionable data and perform many repetitive update, physics, and rendering computations on that data.

One example of this is shown in the thesis "Scalable Multi-threaded Game Engines using Transactional Memory" [3]. In this thesis, the author describes a method where the update step of a game engine is broken up into many multithreaded processes through the use of spatial partitioning [3]. An example of this is shown in Figure 2-2. In this example the world is

partitioned into several different sections, and each of those sections is executed on a separate thread.

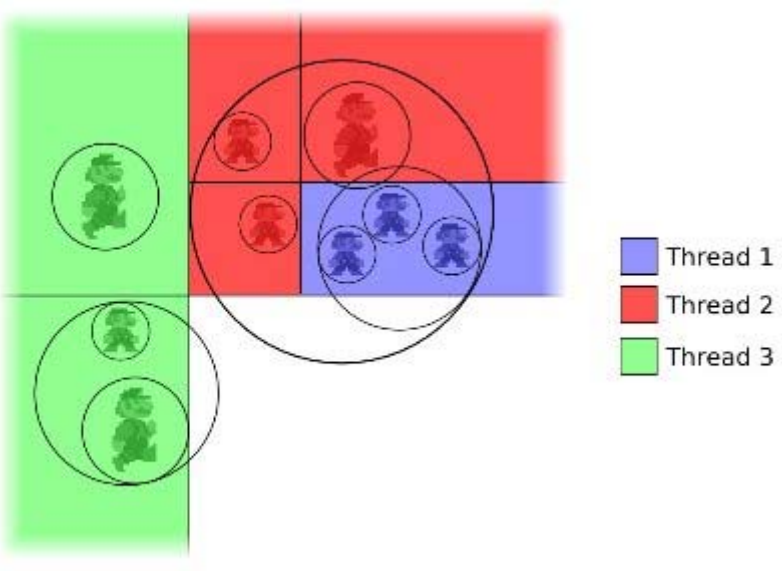


Figure 2-2 An example of data parallelism. [3]

In such a system, the number of threads that the system is to use is defined at runtime and the world is broken up using a spatial partitioning method such as a quadtree or kd-tree. Each of the spatial sections of the world and all of its contained game entities are then given to a thread to be processed. In this way, multithreading is accomplished and is limited not by the number of reasonable tasks that the processing of the game can be broken down into, such as in task parallelism, but instead is only limited by the number of the cores and the complexity of the scene.

Because of their greater scalability and applicability, data-parallel game engines are the more ideal multi-threading solution for game engines. They do, however, have the great disadvantage of requiring that all of the state data for the game be synchronized between multiple cores or machines. This is a difficult problem, and has had much research done on it in the context of game engines.

2.5 Data Synchronization

In order to have multiple threads or machines working on the same set of data, such as in a data-parallel game engine, that data must be synchronized between all clients. This is a difficult problem, and presents many unique challenges. There have been many techniques explored in this area that seek to provide maximum data coherence while still maximizing performance and not hindering the overall system.

The paper "A Realistic Distributed Interactive Application Testbed for Static and Dynamic Entity State Data Acquisition" [2] proposes that one way to greatly improve the performance of existing data-synchronization methods is through predictive algorithms. In most games today, the local client application will use "dead reckoning", or a best guess, to try to guess what the future position of objects will be in absence of information from the server. The author proposes another method which would interpolate the information on the client by using a predictive algorithm that would anticipate what the future state of objects in simulation would be based on their prior behavior.

This method is interesting, and if implemented well it could greatly reduce the amount of information needed to be sent between different clients in order to synchronize data. Unfortunately, this thesis only describes a hypothesis and preliminary work on the subject and does not advance a fully developed implementation.

The thesis "An Efficient Synchronization Mechanism for Mirrored Game Architectures" describes a more fully-fleshed method for synchronizing the data between multiple game clients and servers. This is an optimistic method that lets all clients execute instructions until a conflict between the states of two clients is detected. When such a conflict is detected, the state of the trailing client is rolled back and brought back into sync with the leading client [1].

Such a system is very useful because it allows all clients to run optimistically and transfer the minimum amount of data between each other. Only when a conflict is detected does full

synchronization take place, increasing the overall efficiency of the system. This kind of algorithm could be very useful in the implementation of a data-parallel game engine in order to minimize the overhead of keeping multiple clients synchronized and maximize the gains of parallel processing.

3 SDSE: An Implicit Processing Model

SDSE, or the Sinking Duck Simulation Engine, was our initial implementation of a parallel simulation system for this project. It was derived from the Entity-Variable-Aspect system developed by Chris Miles [9], with additional features to allow for processing in a networked parallel environment. These modifications took the aspect processing model already in the system and load-balanced that processing across a heterogeneous network of computers using a greedy algorithm. Communication was automatically handled so that the variables of entities and messages appeared to be handled locally, but were actually synchronized across the network when needed.

3.1 EVA

The original EVA system, as developed by Chris Miles [9], was a way of organizing a game or simulation. It was developed at the Evolutionary Computing Systems Lab at the University of Nevada, Reno for the naval training simulation and Awe research platform Lagoon. The three basic components in this model are entities, variables, and aspects.

An entity is the basic object of a simulation, and represents something in the virtual world. This could be anything from a building to a car to a bullet. An entity contains a collection of variables and a collection of aspects. The variables represent the state of the entity for such things as size, color, and velocity. The aspects represent different behaviors or natures that can be assigned to an entity, such as making it a physical object, making it graphically represented, making it generate sounds, or having it react intelligently to its surroundings. On the technical side, aspects can be thought of as pieces of code that can be attached to entities. There are many hooks that an aspect provides for code to be attached to, including update calls based on timers, initialization and destruction of the entity, variable updates, and messages that are passed between

entities in simulation. The composition relationship between these three aspects is shown in Figure 3-1:

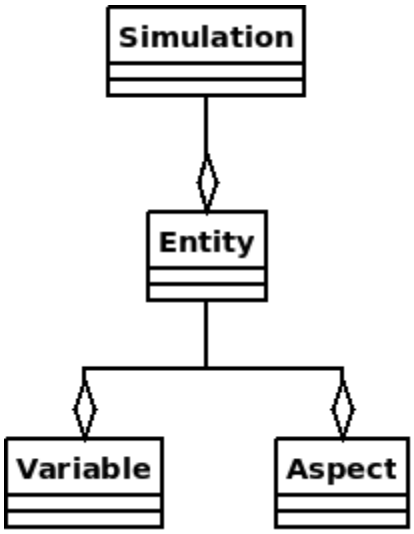


Figure 3-1 The composition model of EVA [9].

The power of the EVA processing model comes from the decoupled way that different aspects can be put together. Since aspects communicate through variables, they are agnostic as to where those variables are coming from or going to. As an example of this, consider a design to represent a ship. In this design, we have a basic physical aspect for the ship entity. This physical aspect receives data via the throttle and rudder variables of the ship, and outputs variables in the form of velocity, rotation, and position. Above this physical aspect, we can have two alternatives for how to create the input variables of throttle and rudder. One alternative here would be to create an aspect that looks at the state of an external hardware interface and sets the control values directly, allowing the user to drive the ship. Another option would be to create an Awe aspect that inspects the state of the world, uses a set of goals, and generates the appropriate rudder and throttle values. Because of the modular nature of aspect processing, the physical part of the ship entity doesn't need to be recreated for each ship implementation. This control loop is shown in Figure 3-2:

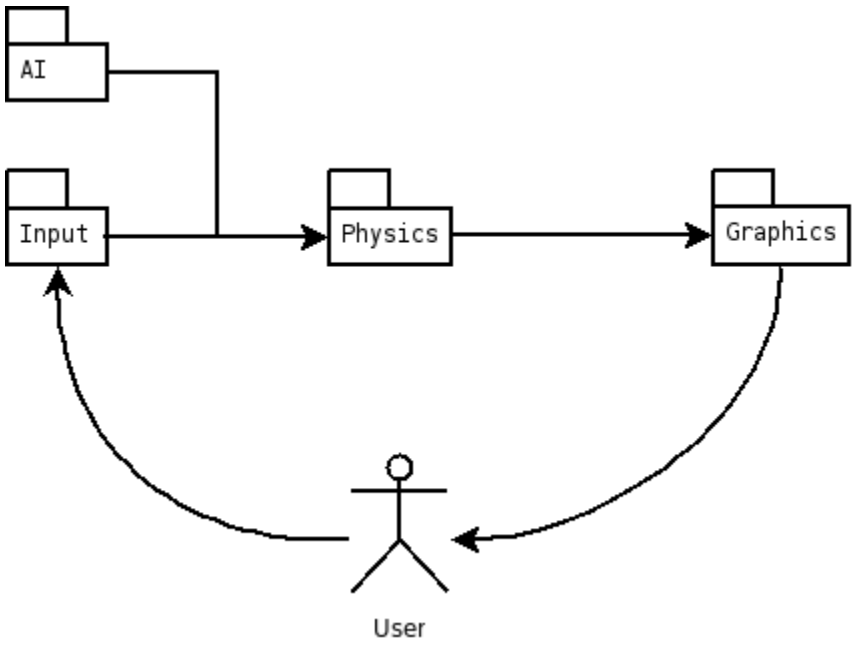


Figure 3-2 The simple interaction model of EVA.

3.2 EVA as AOP

AOP, or aspect oriented programming, is a programming paradigm that is derived from the visitor design pattern. The visitor design pattern is an object oriented pattern with data objects built to accept generalized visitor classes [17]. This can be a very useful way to decouple data and the operations on that data, because visitor classes can be changed without requiring any changes in data classes.

AOP takes the visitor one step further by describing a method of programming where classes can have some functionality defined by aspects that are attached to those objects [19]. By using these generalized aspects, it is possible to maximize reusability and maintainability.

For an example of AOP, consider a financial application with many different classes to represent things such as customers, accounts, and transactions. All of these classes need to implement logging functionality to record their actions.

One option for implementing this system would be to implement the functionality of logging for each object inside of that object. This is the obvious design and will work well

initially. However, the downside of this design will become apparent when a change needs to be made to the logging system. In order to change the logging system, it is necessary to change every class that features logging functionality. This represents a huge cost of development time.

Another option for implementing this system is to implement the functionality of logging in a generalized logging aspect. This aspect can then be attached to many different objects without modifying those objects. This is a good design because if there are changes necessary for the logging system, only the logging aspect needs to be changed.

AOP is not cited as a direct inspiration for EVA in [9], but the two methods have many common characteristics. Each system seeks to maximize modularity and maintainability by dynamically mapping aspects to objects. In this way, EVA can be seen as an application of AOP to games.

3.3 EVA as a System of Systems

In [15], Plummer discusses an architecture for computer games known as a system of systems. In this architecture, the different systems that make up a game such as graphics, physics, and input, are separated into strongly decoupled components. Maintainability is greatly improved through the use of this architecture, because components can be added and removed without affecting each other.

Consider the example of a game that uses one library, such as Bullet, for physics. If the developer later decides to use a different physics library, this can represent a lot of work in dealing with repairing dependencies broken by the change. If, on the other hand, the developer has used a system of systems architecture, the maintenance development cost and time are greatly reduced because there is no need to modify any system but the one being changed.

Consider a group of aspects in EVA that does one task, such as graphics rendering. This group of aspects is analogous to a system in Plummer's system of systems architecture. To change how any task like graphics or physics is accomplished, it is possible to change out the

group of aspects that accomplishes that task without affecting any other systems. In this way, EVA can be seen as one implementation of the system of systems architecture.

In addition to the benefit of modularity, Plummer notes that another benefit of this system of systems architecture is the possibility of parallel execution. He notes that if it is possible to analyze the dependencies between the different systems, it will be possible to dynamically separate the execution of the separate systems into different threads. In the next section, the SDSE system is describe which does just this. SDSE analyzes and exploits the dependencies between aspects in EVA to speed up execution through parallel execution.

3.4 SDSE-EVA

The original design of EVA was modified in SDSE in order to work in parallel. The basic design lends itself fairly well to this adaptation, and only a few modifications were required. First, the aspects of the system, which represent the central processing of game logic, were set up to be distributed over the network. In order to accomplish this several other changes were required. The state of the game, in the form of variables, had to be synchronized on demand across the network. Also, any messaging had to distributed transparently across the network.

The first part of the changes made to EVA for SDSE, the distribution of aspects, is the most intuitive part. If we can distribute the processing of these aspects, then we have distributed the processing of the simulation logic. In order to facilitate this distribution, a server was developed that kept track of all working clients in the network and which aspects each client was capable of running. This heterogeneous nature of the network was useful for many applications where the machines doing processing were heterogeneous such as a headless cluster running Awe for a simulation with graphics being run on a desktop head. When any client on the network requested an aspect to be run on an object, a message would be sent to the server requesting that aspect to be run. The server would then look through the registered clients and their current work

loads and determine the most idle client to run the aspect. This accomplished a naive greedy load balancing which proved adequate but could be improved.

The overall architecture of this system is shown in Figure 3-3:

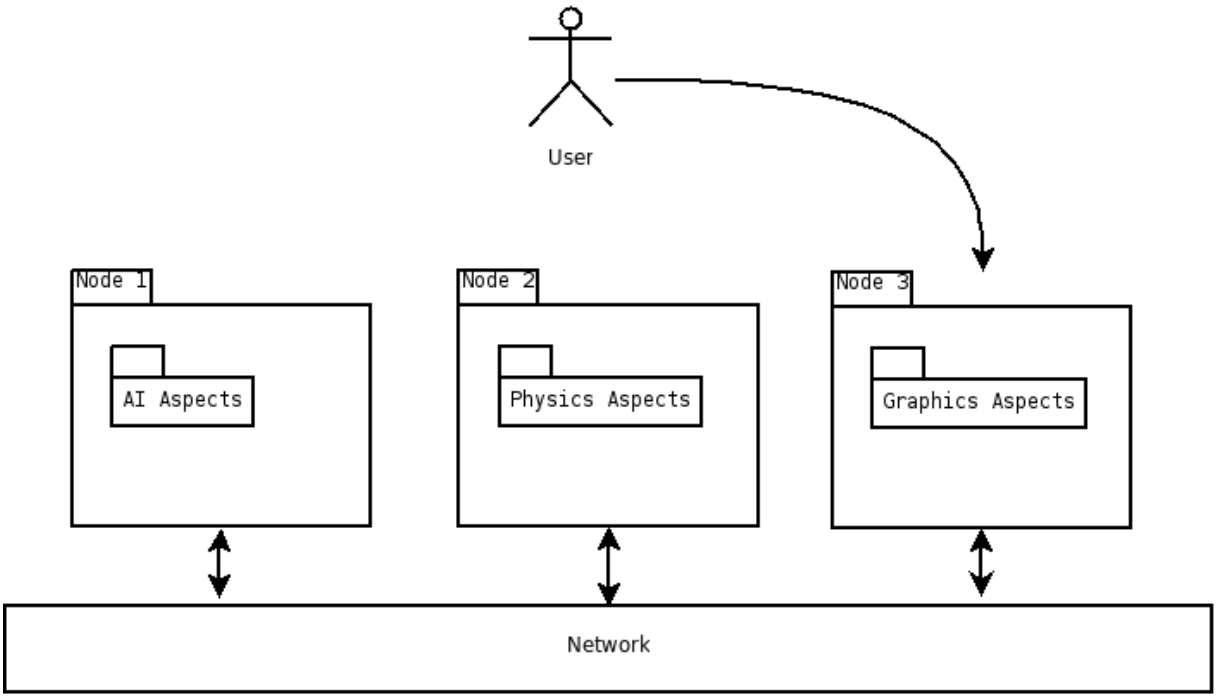


Figure 3-3 The modified interaction model of SDSE-EVA.

Consider that in the original EVA design, communication between aspects happens through the variables in entities. In order for aspects running on different machines across the network to communicate, the variables that they read and set must also be synchronized. Several methods were developed to efficiently deal with this system, and the solution that was eventually arrived at was an on-demand system because this system minimalized communication through the high-cost sockets.

This on-demand variable synchronization system operated on the principal that a client only needed to be notified about a variable's value changing if any aspects running on that client read that variable. In order to implement this, the server kept a list of requested variables. These requested variables were those variables that had been read on each client. The flow of work on a

client was that it was always notified about the initial value of a variable, and on the first read of that variable it would get that initial value. On that first read, the server would be notified that the client was interested in that variable, and all subsequent variable updates would be propagated to that client. This system meant that variables were sometimes out of date on initial reads, but that most of the time they were up to date and that only important information was generating traffic on the network.

Another optimization to the variable synchronization was similar to the first, but applied to clients setting variables. The first time that a client set a variable, that value would be propagated across the network. The client would then wait for that variable to be requested before it sent update values to the server again. This meant that when another client requested updates for that variable by reading it, the server would notify the setting client that the variable should be sent when locally updated. By combining both of these variable synchronization optimizations, both traffic from clients notifying the server of variable updates and traffic from the server notifying clients of variable updates was drastically reduced.

In SDSE, an asynchronous messaging system and a message publish-subscribe system were also added. These features were added because in addition to the synchronous communication offered by variables, it is also sometimes necessary to have asynchronous communication for things such as user input and network communication. Also, when working with different aspects on different machines across a network, it is impossible to have one aspect simply call a function on another, since the target aspect may not be on the same machine. Instead, the messaging system can be used to call a managed aspect hook that may be anywhere on the network.

3.5 Other SDSE Features

So far the description of SDSE has been at the high level of the system architecture, but there were many systems developed within it to develop it into something approaching a workable game engine. These included a scripting interface, graphics, physics, input, and a GUI.

The scripting interface of SDSE was developed using Boost Python [21] and presented a rich python environment where the user could easily control all aspects of the engine. This interface was controlled such that all functionality was exposed, but in such a way so to not confuse script developers with the higher level of functionality offered by the core engine.

The graphics of the system used Ogre [22]. Ogre, the Object Oriented Graphics Rendering Engine, is an open source graphics engine that provides many cutting edge graphics capabilities such as scene-management, shaders, post-effects, etc. It was incorporated using the aspect system, with aspects developed for a graphics server and various scene objects.

The purpose of the graphics server is to provide access to the central Ogre objects. These include the overall scene, global settings such as light and fog, graphics resource management, etc. When a graphics aspect is created, it looks for the graphics server in order to create the underlying Ogre object that it will use such as a model or particle effect.

There are many graphics aspects implemented, including different primitive shapes such as spheres, boxes, and planes, as well as more complex objects such as meshes, lights, and particle effects. All of these aspects expose the main settings of each of their underlying graphics objects to the game system through variables. This allows settings such as the filename a mesh is loaded from or the material of a sphere to be set from files or scripts. These aspects also implement listener functionality that listens to many values such as position, orientation, and scale and updates those values immediately to the underlying graphics framework. This means that these aspects are consumers, that is, they sit and listen for values to be set on their entity, and use those values to do work.

The end result of this system is that when these aspects are created on objects, any other aspect, file, or script can come along and set the properties for the graphics and have those changes reflected to the display on screen. This is very useful because it decouples the graphics from having to know where the values are coming from, and allows anything to feed the graphics information from a physics system to an Awe system to a user at an interactive python prompt.

The physics system in SDSE is implemented using Bullet Physics [23]. This is a cutting-edge physics system that can simulate many different bodies, joints, and constraints. The wrapper used to encapsulate this library is similar to the wrapper used for Ogre: it consists of a physics server and body aspects.

The physics server is the central connection to the bullet physics library, and holds all the bullet-specific objects necessary for managing the physics world. This includes the collision and dynamics world, and all of their settings including gravity and calculation constants. The physics server also monitors collisions between bodies and broadcasts a message when one occurs. This is all implemented as an aspect.

There are several physics bodies implemented that connect to the physics server. The physics bodies connect to the physics world to create underlying constructs, and connect through the variable and message systems to allow control over various properties such as size and mass. Each time tick, all physics bodies are updated through bullet, which then sends the updated values for position and rotation for each body. These values are propagated up through to the engine using physics listeners known as MotionStates. Each body aspect has one of these motionstates registered into the physics server for its body, and sets the appropriate variables in the object when that motion state changes. These values can then be used by any other aspects, such as graphics.

3.6 Application Programmer Interface

With all of these modifications, it was possible to run an EVA-style simulation in networked parallel. The process of creating a simulation for this system was largely similar to that of creating a simulation for EVA. The user would create an XML file that described entities in the simulation in terms of variables, scripts, and aspects. The system would then take these files and combine them with the structure of the network in order to run the simulation in parallel. If the entire simulation was run on a single node, the networking would be optimized away and the simulation would run in a way very similar to and almost as fast as the original EVA. There are four main components that make up an application in SDSE. These are the aspects, scripts, definitions, and external resources. The basic structure of the definition of a simulation is shown in Figure 3-4.

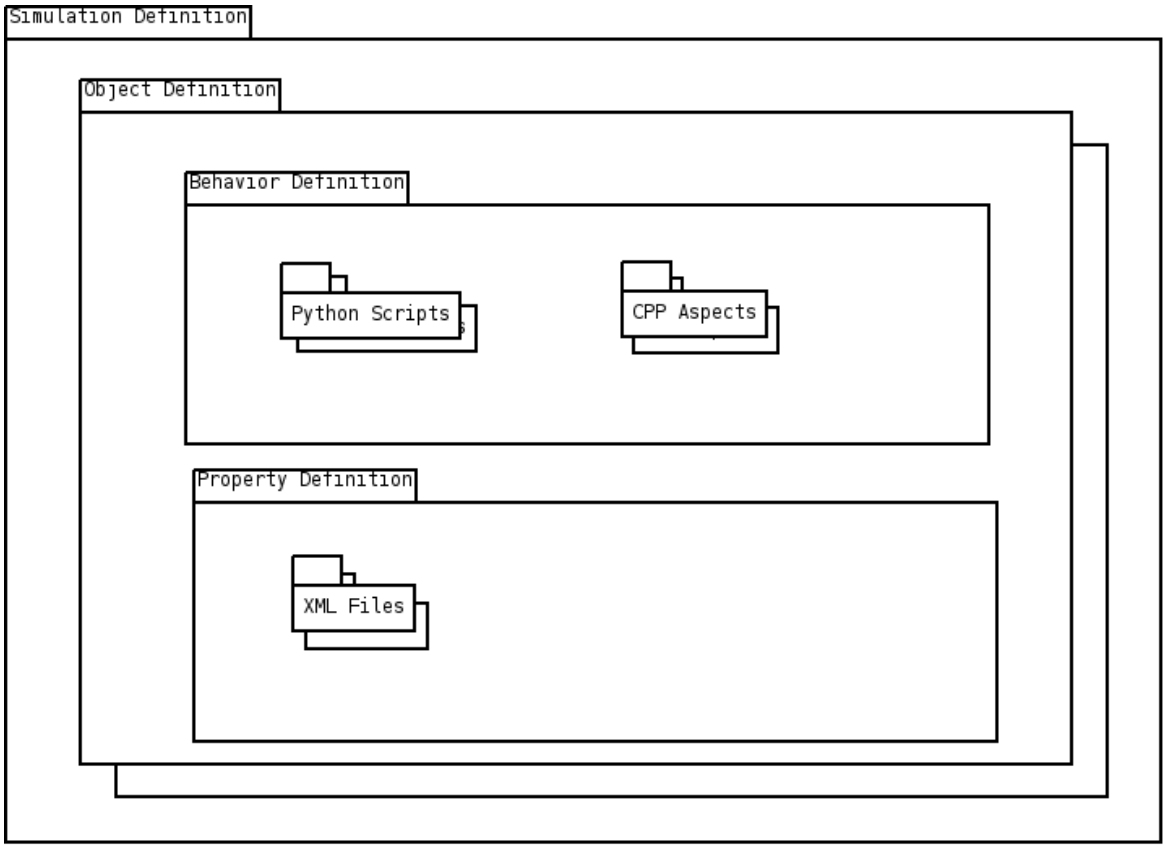


Figure 3-4 The layout of a simulation declaration in SDSE.

Aspects are implemented as C++ classes that inherit from the base SDSE::Aspect class. This base class exposes all the functionality of the system to the developer. This includes basic start and stop methods that are run when the aspect is added and removed from an entity. The developer is also provided with functions to subscribe to variables getting set on the attached entity. This is useful for aspects that consume variables, such as a graphics aspect that would want to update the graphics representation whenever a variable such as position or color is set. Finally, it is possible for aspects to register callbacks for the asynchronous messaging system. This allows application developers to react to events, such as game logic reacting to a button being pressed on the keyboard.

The build environment is constructed such that build meta-tags can be inserted in aspect source files to notify the system of many things such as how the aspect will be registered into the system, what libraries it depends on, etc. At compile time the system discovers all aspects available and registers them into the system, making them available to be added by name from any source including other C++ code, python scripts, or xml files. Aspects are useful for implementing functionality that must be high-performance, or that will be reused frequently.

Scripts fulfill much the same role as aspects in the SDSE system, but in a more portable way. While the addition or modification of an aspect requires a full recompile of the system, a script can be modified without any recompiling. The script environment provides almost all of the functionality that the C++ system does, excluding direct access to aspects in code. The script developer communicates with the rest of the system using variables, asynchronous messaging, and object and aspect management. Scripts are a useful way to define application code without requiring the implementation overhead of implementing a full aspect and with the fast, intuitive interface that python provides.

Aspects and scripts provide behavior for the objects in a simulation, but the objects in that simulation have to be created through data files. This is done through xml files using a modifiable, extensible format. All parts of an object can be described through these files, including aspects, variables, and scripts. These data files represent the simulation itself.

The final part of an application defined for SDSE is external resources. This is a catchall category, and includes everything from graphics files used by the rendering engine to sound files used by the sound engine.

All of these files are discovered at runtime and provided through a file service in the engine. This allows paths to various files to be specified absolutely, relatively, or searched for by name or type. This increases the ease by which applications can be developed and reduces expert knowledge required of the application developer. Through all of these systems, a simulation for

SDSE can be described concisely and intuitively without reducing the expressiveness of the interface.

4 SimQ: An Explicit Processing Model

4.1 Overview

SimQ, short for simulation processing queue, was developed as an improvement on SDSE. The lesson learned from SDSE was that an implicit model for transporting information from one processing unit to another worked but was not optimal. The solution that was found for this problem was to move into a multi-threaded model of processing, with an explicitly structured queue of operations waiting to be processed by each component of the system. By creating this flow of information through the system, it is possible to ensure that each component only acts on an entity when it is required and work is never repeated.

This system is currently not fully developed, but the core scheduling and automatic parallelization system is already in place. In this system, a simulation object represents a complete simulation with all processing components, or services, and all game entities. There is also a many-to-many relationship between services and entities that shows which entities are acted on by which services. Services are structured to accept an incoming stream of entity update packets. These packets tell the service which entity needs to be updated and why. This framework is designed to be extensible so that if a new type of communication or state update is added to the system, services will be able to easily adapt to react to it. The overall structure of this system is shown in Figure 4-1:

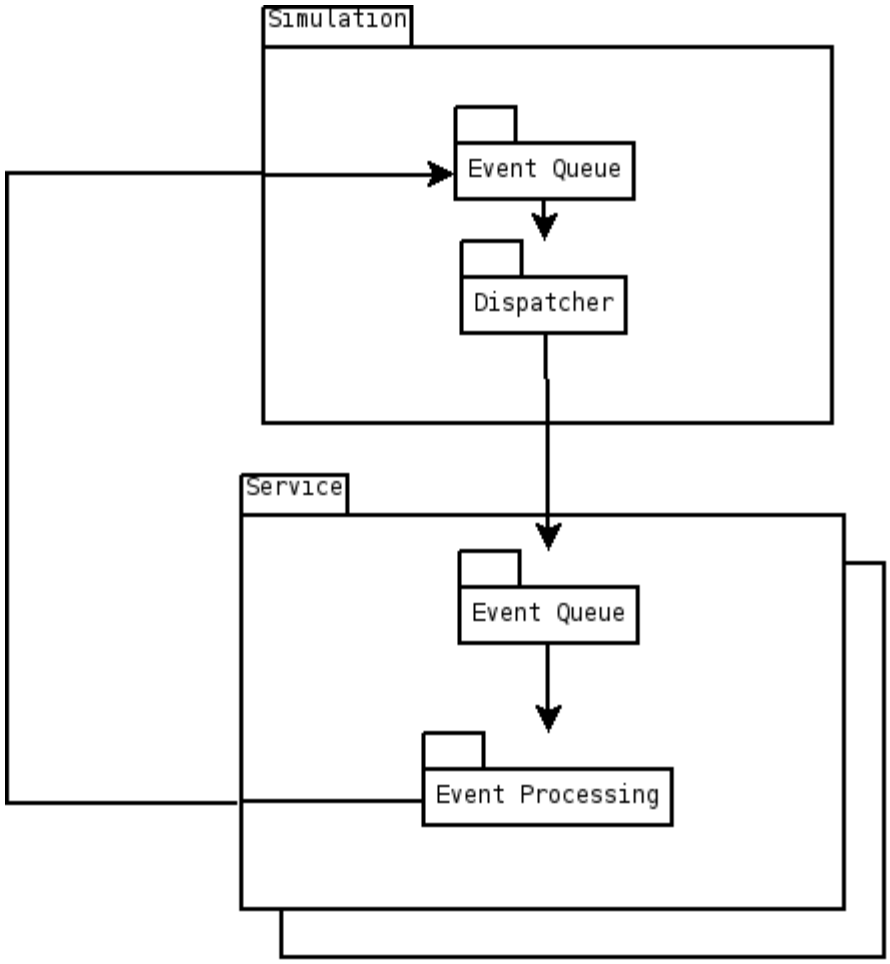


Figure 4-1 The processing model of SimQ.

Services represent components of the system that do various forms of work such as graphics, physics, AI, etc. The primary difference to note between services and SDSE aspects is that while an aspect is attached to and works on a single entity, a service represents a processing filter that entities are constantly passing through. This passing through is a vital part of the parallelization of the system, and allows a pipe-and-filter architecture to be established for the processing of entities.

4.2 Advantages

An example of a simulation built on this system consists of a physics service and a graphics service. The physics service is driven by timer tick messages at a fixed interval, and

produces state update messages that are consumed by the graphics service. This way, the graphics service is only receiving messages when it needs to update its internal information on entities. Each time the physics driving timer fires, an entity is passed through the physics service to be updated and passed to the graphics service. While this first entity is being processed by the graphics service, a second entity can be updated by the physics service. This pipe-and-filter style of parallelization is scalable, and the message passing style of synchronization between services allows the processing path of entities to be explicit and deterministic.

By explicitly designing this flow into the structure of the system, services are prevented from doing redundant work. Consider the case of SDSE-style processing, where the graphics and physics are both allowed to constantly update entities as fast as they can. If our hypothetical physics service is much faster than our graphics service, there are going to be many physics updates that are not propagated to the user for each graphics update. This is wasted processing, because the user doesn't see these updates. Even worse, consider the opposite case where the physics is much slower than the graphics. In this case, the graphics will render the same frame several times for each physics update. This is wasted effort because there is no new information for the graphics to display. In SimQ, this kind of imbalance is prevented by having entities pass through each service before the next service is notified that the entity even needs to be updated.

4.3 Drawbacks

Although there is a gain in performance by using this system over SDSE, there is also a trade off of performance for flexibility. While the application developer can still use the same declarative methods for creating a simulation as SDSE, developers working to create new services or working to adapt existing libraries will have a harder task than the creation of aspects in SDSE.

The first reason for this is that the structure of services in SimQ is less intuitive than the structure of aspects in SDSE. Aspects follow the conceptual model of game creation, allowing a

small amount of code to be added to an object to have that object fill some role. This code is concise, intuitive, and has its own state. On the other hand, when creating a service for SimQ, the developer must implement all functionality associated with that service in one centralized location. It is important to note that this centralization of functionality is not exposed to the end application developer because in both cases the user simply connects aspects or services to entities in the simulation. Only the interface that the service developer implements must be different.

Another reason that service development for SimQ is more difficult than aspect development for SDSE is that SimQ uses a non-standard programming paradigm in the form of raw message passing. It is worth noting that message passing is sometimes used by libraries such as the Message Passing Interface, or MPI, but this technology is not often used for game engines [24]. This is not a standard way of developing game engine components, and requires a shift in mind-set. The main thing that could be a problem with this message passing interface is that initialization is always implied, meaning that the service is expected to always be in a valid state. This is very difficult to program to because it vastly limits the number of assumptions that can be made about the initialization order of different entities and variables.

This lack of ordered initialization could be a problem for adapting existing libraries to work in SimQ. Consider a physics engine that requires the world to be created with a set of arguments before any objects are created. It is impossible to know when all arguments for the world have been received, and some will have to be set to default values in order to proceed with object creation requests. This might result in unexpected behavior for the user, and is undesirable. There are possible workarounds for this that might patch some functionality, but the asynchronous nature of the message passing system prevents any universal solution from being found at this time.

Despite these minor disadvantages, SimQ is still a promising architecture and has a potential for pushing simulation performance very far.

5 Experimental Results

5.1 SDSE

SDSE performed well as a standalone game engine because of the shortcut methods implemented to avoid networking when possible. When run across several nodes, however, performance was only average. The test simulation that was used to stress SDSE is described next. This simulation featured a floor plane and many bodies that were all dropped onto the floor in a pile. All physics was processed by a node running Bullet and all graphics were rendered on a node running Ogre. The entire simulation was defined using one xml file to define the world and one python file to define the object dropping behavior. The same definition files could be used to drive either the standalone version of the simulation or the version running in networked parallel. A screenshot from this test is shown in Figure 5-1.

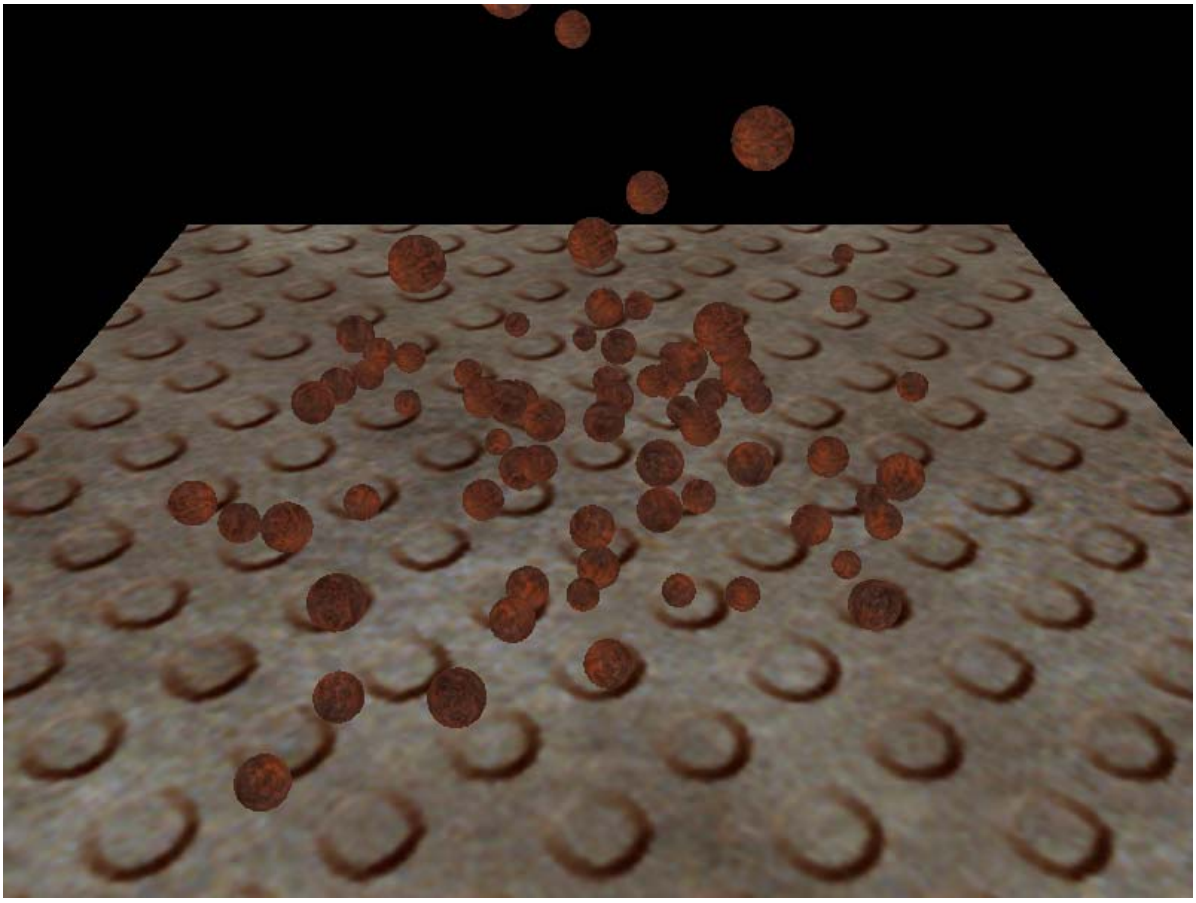


Figure 5-1 A screenshot of the SDSE stress test.

This simulation was tested on a laptop with a dual-core athlon CPU and an nvidia 6800 graphics card. The standalone line represents performance on one core, while the multithreaded performance represents the simulation running on two cores. It is also important to note that the term fps for the parallel graphics and physics nodes refers to the speed at which each of those nodes processes the complete list of entities in the simulation. The actual throughput of the entire system is the fps of the graphics node. The results of these tests are shown in Figure 5-2.

Number of Bodies	10	100	200	300	400
Standalone FPS	952	635	322	187	116
Parallel Graphics FPS	160	61	33	30	25
Parallel Physics FPS	284	155	82	66	41

Figure 5-2 SDSE test framerates.

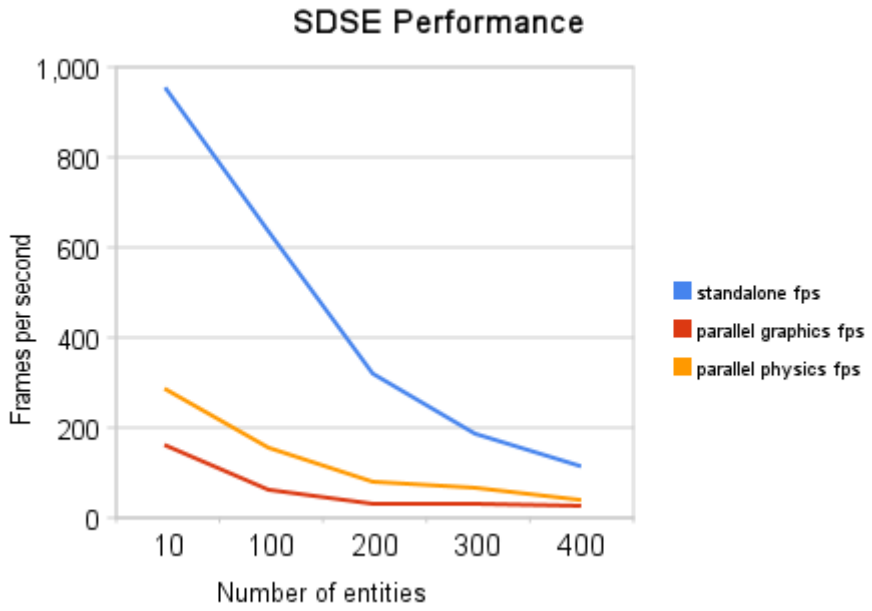


Figure 5-3 A graph of the performance of SDSE

From the data in Figures 5-2 and 5-3, it is possible to draw several conclusions about SDSE's performance. First, it is obvious that the goal of increasing performance by running tasks

in parallel hasn't been accomplished. In fact, running with one node runs with about a 4x speedup over running with two nodes. This is due mostly to communication overhead. This overhead was reduced as much as possible, but in the end it was still a full-time synchronization problem and would never have perfect performance. This explains most of the difference between the standalone and networked performance of the engine, but it is still worthwhile to note that there was a significant difference between the speed of the graphics and physics nodes in the parallel application.

This difference between the performance of the graphics and physics nodes is very telling about the problems in the distribution of work in this engine. Every time the physics node processes an entity, it sends an updated variable packet across the network to the graphics node. Every tick of the engine on the graphics node, the system processes all incoming messages before trying to render out a graphics frame. This processing of input messages might involve much redundant work, as a variable can be sent across the network several times before it is possible for the graphics node to go through its queue of incoming messages. This redundant processing is a significant cause of the reduction of performance in the system.

One obvious solution to this problem would be to only process the latest update to any given variable in the update queue, but unfortunately the asynchronous nature of the processing of the incoming network stream prevents the engine from having any easily-definable point where the queue of variable updates could be optimized to prevent redundant work from being done.

5.2 SimQ

The two problems of communication overhead and worker synchronization were addressed with the design of SimQ. This new system moves into a shared memory multithreaded system, greatly reducing the cost of communication overhead. It also strives to alleviate the problem of

worker synchronization by creating explicit queues of pending tasks for each service that can be optimized without affecting the processing done by the service.

The communication of SDSE was done through UDP sockets. This is very slow, and SimQ improves on it by moving to a more standard managed multithreaded environment. Synchronization still has some performance cost because of the necessity to synchronize access to entities and variables through locks, but it is possible to optimize this still further through the use of many cutting-edge technologies. This is a large gain of performance for SimQ over SDSE.

Another way that SimQ improves its performance over SDSE is by having explicit task queues for each service. These queues represent all of the pending work for a given service, and can be optimized to prevent redundant work in a way that wasn't possible in SDSE due to its asynchronous architecture.

Both of these optimizations were designed to specifically address the problems evident in SDSE, and were shown to be effective in the test case shown below. This test was run using a stubs for a producer-consumer model of parallel processing similar to the graphics and physics nodes used to test SDSE. In the test, there are a number of entities that are processed by a producer service as fast as possible. This producer service sets a variable on each entity which triggers a setVar entity update event which is then queued and processed by a consumer service. The consumer service then gets the set value from the entity. Each processing step in the producer and consumer services takes 0.1 seconds. This means that for n entities, there is a baseline linear programming runtime of n * 0.1 * 2 seconds. Keep in mind that the system used to generate these tests had all of the high-cost core features of the engine, including task and variable synchronization. As Figure 5-4 shows, SimQ showed a nearly linear speedup over sequential execution. The simulation is run on three hardware threads, with one for the producer, one for the consumer, and one for the game server. The experiment was run on two cores, and the average overhead of parallel is 5%.

Number of Entities	100	200	300	400	500
Single Threaded Time	2	4	6	8	10
Multithreaded Time	1.04	2.09	3.14	4.24	5.33

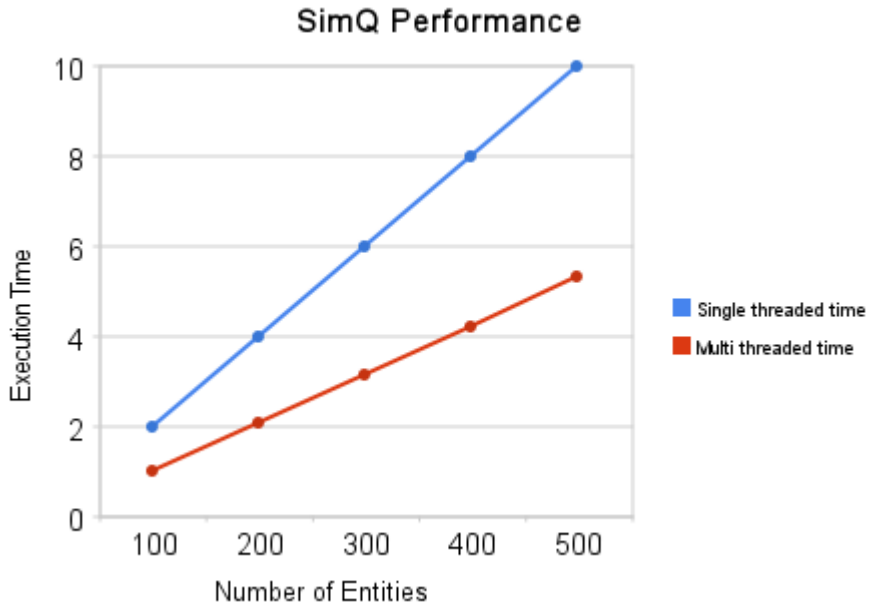


Figure 5-4 SimQ Performance

6 Conclusions and Future Work

6.1 Conclusions

SimQ performed much better than SDSE. This was because of the synchronization and communication overhead of a networked engine. The improved performance of SimQ, however, does come at a cost. This cost is the increased difficulty of developing services for SimQ as compared to SDSE.

In SDSE it is very easy to develop services for features such as graphics and physics. This is because a processing node in SDSE is run in almost exactly the same way as a standalone application. A programmer developing a service for this system can assume that they are developing for a standalone application and still have that service function perform perfectly well in the parallelized system of SDSE.

In SimQ, on the other hand, the service developer must be very aware of the parallel platform for which they are developing. The service must be designed to respond to asynchronous events coming from the overall system and from other services. At the surface level, this seems similar to the interface of SDSE. After all, in both systems services are developed to respond to timer ticks, broadcast events, and variable updates. The key difference, however, is that there are no guarantees for packet ordering and delivery in SimQ.

A service in SimQ, such as a physics service, might look for several variables in each entity to determine how to process that entity. These variables could include things like body shape, mass, momentum, etc. Many external physics libraries require that this information be provided all at once in order that the library can create appropriate constructs. In SimQ, however, there is no time when it is possible to say that all data has arrived, and the service must be written

in a sufficiently flexible way that any data can arrive at any time and be processed coherently. This is a daunting task to face a service developer with.

The experience of developing SDSE and SimQ has shown the importance of choosing an appropriate architecture for the problem at hand. The architecture used for SDSE was a modified version of the AOP-inspired EVA system, while the architecture used by SimQ was more complex. The logical structure that SimQ provides to programmers and application developers is very similar to that of SDSE, and is expressed in terms of entities, variables, and aspects. Behind the scenes, however, this structure is translated into a pipe-and-filter style processing which is handled by the entity update queues in each service.

This mixed architecture has proven to be effective in SimQ so far, and holds promise for future development. By creating a system that uses an intuitive simulation representation and a powerful background processing architecture, this system is poised to become a very powerful and full featured game engine.

As we can see from the examples of SDSE and SimQ, there exists a tradeoff between ease-of-development of services and service synchronization in parallel game engines. When writing a naive system such as SDSE, it is possible to provide an environment that is transparent and effectively the same as a procedural application from the service developer's point of view. On the other hand, it is possible to write a fairly specialized system such as SimQ that goes to great lengths to provide an environment where the speedup from parallel execution will be maximized, but where the development of services will have to make significant concessions to the overall system architecture in order to work properly.

This tradeoff between ease of development and performance must be carefully considered when considering the use of or developing a parallel game engine. For example, if a project requires the use of existing physics and graphics libraries, SimQ will not be a good choice because of the difficulty in adapting existing libraries to work in the system.

6.2 Future Work

In the future we would like to develop SimQ into a full-fledged game engine. As the previous chapter concluded, this would be a difficult task because of the interface that services must be developed to work in the engine. we don't believe that these are insurmountable challenges, however, and would like to move forward to try to get a fully featured game engine up and working.

Since SimQ is implemented in C#, we plan on working on adapting it to the XNA framework. XNA is a proven platform and will open the door for the possibility of applying these techniques to the Xbox platform. XNA provides functionality for graphics, sound, and input. we will also be incorporating physics via the JigLibX library.

Looking into the distant future, we would like to create a WYSIWYG editor in the style of the Unity engine. This editor would be built in the game engine, and would allow a drag-and-drop style of game creation. This would relatively straightforward thanks to the declarative nature of simulation creation using this engine.

With these improvements, we believe that SimQ could become a powerful force for the parallelization of games with minimum effort on the part of application developers. This could be a powerful tool for development on modern console platforms which feature multiple processors and greatly improve game performance for end users.

References

[1] E. Cronin, B. Filstrup, A. Kurc, S. Jamin, "An Efficient Synchronization Mechanism for Mirrored Game Architectures," in *Architectures, Multimedia Tools and Applications*, New York: ACM, 2002, pp. 67-73.

[2] D. Marshall, D. Delaney, A. Mccoy, S. McLoone, T. Ward, "A Realistic Distributed Interactive Application Testbed for Static and Dynamic Entity State Data Acquisition," in *Proc. ISSC*, 2004.

[3] M. Silverman. Scalable Multithreaded Game Engines Using Transactional Memory. Undergraduate thesis, University of Rochester, 2008.

[4] J. Tulip, J. Bekkema, K. Nesbitt, "Multi-threaded Game Engine Design," in *Proc. 3rd Australasian Conference on Interactive Entertainment*, 2006, pp. 9-14.

[5] N. Vikram. A Game Engine on Distributed Systems using CORBA. Dissertation, Birla Institute of Technology & Science, 2005.

[6] M. Masuch, L. Nacke, "Power and Peril of Teaching Game Programming," in *Proc. International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004, pp. 347-351.

[7] M. Lewis, J. Jacobson, "Game Engines in Scientific Research," in *Communications of the ACM Volume 45 Issue 1*, 2002.

[8] R. Chabukswar, A. Lake, M. Lee, "Multi-threaded Rendering and Physics Simulation," Intel Software Network, 2006.

[9] C. Miles. Co-Evolving Real Time Strategy Game Players. Dissertation, University of Nevada, Reno, 2007.

[10] V. Monkkonen. (2006). Multithreaded Game Engine Architectures. Gamasutra. [Online]. Available: http://www.gamasutra.com/features/20060906/monkkonen_01.shtml

[11] C. Magerkurth, T. Engelke, D. Grollman, "A Component Based Architecture for Distributed, Pervasive Gaming Applications," in *Proc. ACM SIGCHI*, 2006.

[12] J. Anderws, N. Baker, "Xbox 360 System Architecture," in *IEEE Micro Volume 26 Issue 2*, 2006, pp. 25-37.

[13] A. Rhalibi, D. England, S. Costa, "Game Engineering for a Multiprocessor Architecture," in *Proc. DiGRA*, 2005.

[14] E. Anderson, S. Engel, P. Comninos, L. McLoughlin, "The Case for Research in Game Engine Architecture," in *Proc. 2008 Conference on Future Play: Research, Play, Share*, 2008, pp. 228-231.

[15] J. Plummer. A Flexible and Expandable Architecture for Computer Games. Master's Thesis, Arizona State University, 2004.

[16] J. Blow, "Game Development: Harder Than You Think," in *ACM Queue Volume 1 Issue 10*, 2004, pp. 28-37.

[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994.

[18] E. Folmer, "Component Based Game Development," In *Proc. 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, 2007, pp. 65-73.

[19] R. Walker, E. Baniassad, G. Murphy, "An Initial Assessment of Aspect-Oriented Programming," In *Proc. 1999 International Conference on Software Engineering*, 1999, pp. 120-130.

[20] Overview of CORBA, http://www.cs.wustl.edu/~schmidt/corba-overview.html, Last accessed: May 07, 2009.

[21] Boost Python, http://www.boost.org/doc/libs/1_39_0/libs/python/doc/index.html, Last accessed: May 07, 2009.
[22] Ogre Graphics, http://www.ogre3d.org, Last accessed: May 07, 2009.
[23] Bullet Physics, http://www.bulletphysics.org, Last accessed: May 07, 2009.
[24] Message Passing Interface, http://www.mcs.anl.gov/research/projects/mpi/, Last accessed: May 07, 2009.