Complex systems based high-level architecture for massively multiplayer games

3 authors:

Viknashvaran Narayanasamy
The DigiPen Institute of Technology
5 PUBLICATIONS **92** CITATIONS
SEE PROFILE

Kok Wai Wong
Murdoch University
277 PUBLICATIONS **4,407** CITATIONS
SEE PROFILE

Chun Che Fung
Murdoch University
308 PUBLICATIONS **3,496** CITATIONS
SEE PROFILE

Complex Systems based High-level Architecture for Massively Multiplayer Games

Viknashvaran Narayanasamy, Kok-Wai Wong, Chun Che Fung,
Murdoch University
{v.narayanasamy|k.wong|lanceccfung}@murdoch.edu.au

Designing exciting massively multiplayer (MMP) games increasingly involves a greater level of complexity that grows exponentially along with rising expectations from game players. This is mainly due to the massively connected nature of MMP games that allow thousands of players to play the game online simultaneously. Current game architectures do not extend well to MMP games as they are based on pre-deterministic behavior. This gem introduces an architecture based on the emergent properties of Complex Systems to minimize deterministic behavior in MMP games. A bottom-up design approach that relies on heterogeneous agents, self-organization, heavy interaction, feedback and emergence within game objects was used to design a Multi-Tiered architecture that will scale well for next generation MMP games.
As the entire architecture is extensive and incorporates a number of common features found in other game architectures, this gem will only highlight the unique features that this architecture introduces. This gem will take a software engineering approach to provide a High-Level abstraction of the architecture that will enable anyone to implement in their MMP game.

Complex Systems & Emergence

Complex Systems are highly structured systems that have a number of interacting elements that are organized into structures that vary in depth and size. These structures are subjected to processes of change that cannot be described by a single rule or reduced to only one level of explanation [Kirschbaum98]. Complex Systems were originally intended to study the combinatorial problems of parasitism, symbiosis, reproduction, genetics, mitosis, and survival-of-the-fittest in nature and biology [Odell02]. Systems that can be studied as Complex Systems have a number of identifying characteristics, namely self-organizing structures, non-linear relationships, order/chaos dynamics and emergent properties [Kirschbaum98]. Contributors to the Wikipedia [Wikipedia05], have noted that Complex Systems can also be distinguished by the presence of feedback loops, open environments and indeterminate system boundaries.
Massively Multiplayer games exhibit many of the properties and behavior that can be found in industrial applications of Complex Systems. This can be attributed to the highly interactive environment with thousands of human and AI based Non-Playable Characters NPCs) ; large persistent worlds; non-deterministic number of game states and multiple unpredictable inputs from human players from different socio-economical, ethnic and cultural backgrounds.
The game architecture based on Complex Systems that is presented in this gem facilitates the incorporation of some of the desirable features of Massively Multiplayer games that have not been possible in recent years. The architecture facilitates the creation of truly infinite worlds that exhibit emergent behavior. The unique emergent properties of the

architecture enables actions that were not originally intended in the original game design to emerge throughout the lifetime of the game. These actions introduce a level of unpredictability and randomness in game behavior that is paramount for the next generation of Massively Multiplayer Games.

Multi-Tiered Architecture

The Complex Systems based Multi-Tiered Architecture consists of four unique tiers: the Environment Tier, Object Tier, Agent Tier and the Overseer Tier. Traditional multi-tiered and layered architectures exhibit orthogonality between the layers or tiers to reduce coupling between the various layers/tiers. Orthogonality in this architecture is present between the Environment Tier and the other tiers. This was done to ensure that the stability of the environment is independent of the evolving nature of the other tiers. On the other hand, object, agent and overseer tiers were designed to work synergistically and cohesively together to allow for increased interactivity and emergence within game objects.

The Object Tier consists of the component and composite layers. Emergence starts in the component layer of the Object Tier and propagates upwards through the interaction and aggregation of component objects. The Agent Tier represents all game objects that have higher level functions in the game. This includes the NPCs, human players and other game objects that exhibit a significant-level of artificial intelligence. The Object Tier and Agent tier enables designers to make use of simple game rules for component objects to produce organized and structured high-level behavior. The Overseer Tier is the last and most significant tier in the architecture. It makes use of agents to implement multiple artificial game governors that implement policy-based control that have varying influence over all game objects in the game environment.

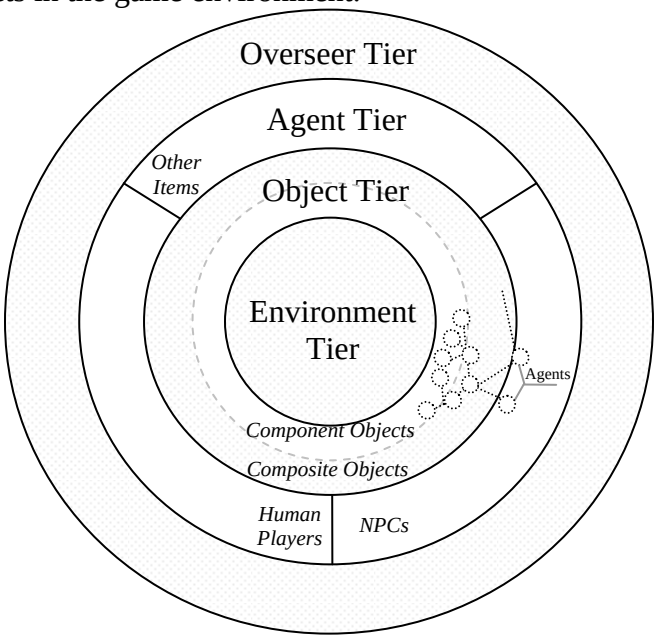

Figure 1. The Multi-Tiered Architecture

Environment Tier

The game environment defines the global properties of the virtual game environment being simulated. The Environment Tier includes these global properties as well as an implementation of the game engine. The *game engine* in the Environment Tier consists of the simulation, graphics, audio, physics and AI engines; the event and input handlers; and the resource and hardware abstraction layers. However, unlike other game architectures, direct access to the game engine resources and game data resources are disallowed. This is done to allow the objects (see *Object Tier*) in the MMP game to run on different client and server machines simultaneously without client-specific knowledge of the platform it is running on. This also allows the game to be implemented on different platforms (i.e. console, PC, web, mobile, etc.), as only the implementation of the Environment Tier has to change for each platform. The Client Machine downloads only a relevant subset of objects required for the client's view of the game environment and maintains the object states to be consistent with the server. The Client's Environment Tier communicates with the Server's Environment Tier only to retrieve and update game data (i.e. Level, Graphics, etc.) during the operation of the MMP game. Figure 2 illustrates the operation of the Client/Server architecture.

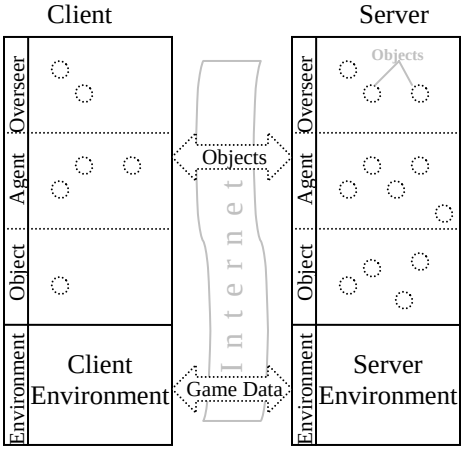

Figure 2. Client/Server MMP Game Architecture

The game objects in the other Tiers can interact with the Environment Tier only through the Event Handler, AI Engine and the Resource Abstraction Layer, as shown in Figure 3. The *AI Engine* implements a number of Overseers (see *Overseer Tier*) to implement an essential set of rules to define the capabilities and the limitation of the client-specific environment. It also implements an essential set of game rules for in-game economics, physics, environmental boundaries, etc. The *Input Handler* retrieves user inputs and generates events to let the game engine handle the inputs. Unlike conventional game architectures the *Physics Engine* is placed at the same functional level as the other engines that interface with hardware to reflect the upcoming trend of off loading physics computation to dedicated PPUs (Physics Processing Units) [Ageia05]. The *Graphics Engine*, *Audio Engine* and *Game Data Module* mirror the functionality of their counterparts' in common game architectures, so details of these functional components will not be described. The *Event Handler* works hand in hand with the Simulation Engine to translate object and user events to admissible events for the Simulation Engine. The Event Handler also routes events as and when they happen to objects that have subscribed to receive specific event types.

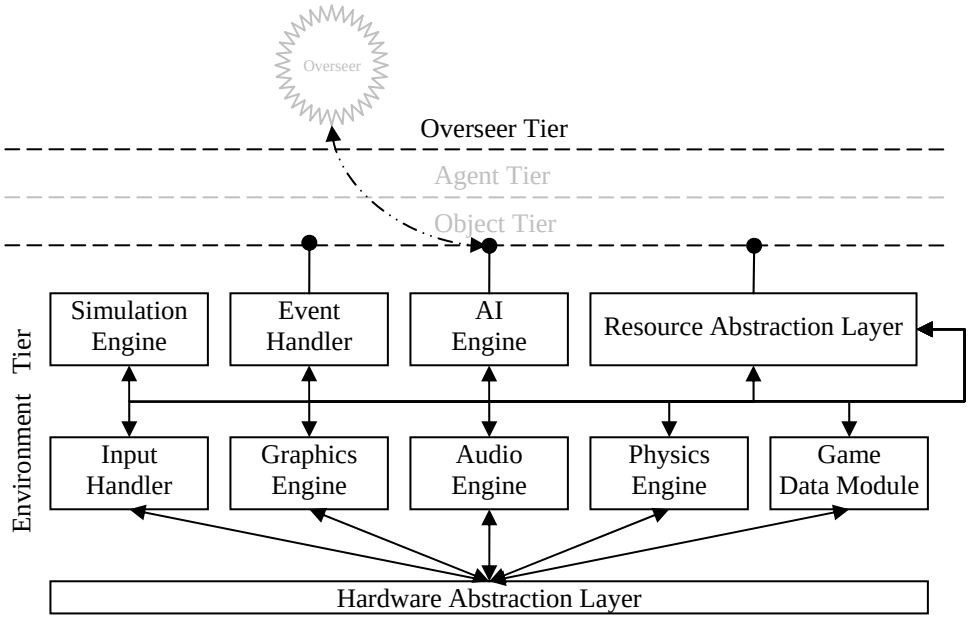

Figure 3. Environment Tier

The Simulation Engine is synonymous to a game loop. It makes use of the Discrete-Event Simulation (DES) Model [Garcia 04]. The DES model was originally designed for executing simulations in complex systems, where the order of execution of events is unknown and unpredictable due to the large number of objects present. In such a scenario, the continuous simulation model used in the majority of games that relies on the sequential polling and execution of events does not preserve the causality of the system. As the simulation processes in a Discrete-Event Simulator are executed the moment they happen, the time order execution of events is preserved. This immediate execution of events also increases the sensitivity of the system's responses to simulation events.

As shown in Figure 4, both the graphics and simulation processes are simultaneously executed. This allows the graphics rendering rate to be independent of the simulation speed. This is especially realizable in the latest generation of gaming hardware architectures that support parallelism.

Events generated by the user and events generated by the program will be queued to be processed. When there are no events queued, the system will go into sleep mode until it is preempted by new events in the queue. Dynamic Simulations that require constant updates can generate periodic events with the help of the system clock to perform updates at a user-definable frequency that is independent of other simulation processes and the rendering sub-system. i.e. simulation of a person walking constantly. The simulation process is ordered and prioritization of events can occur with the aid of the queue.

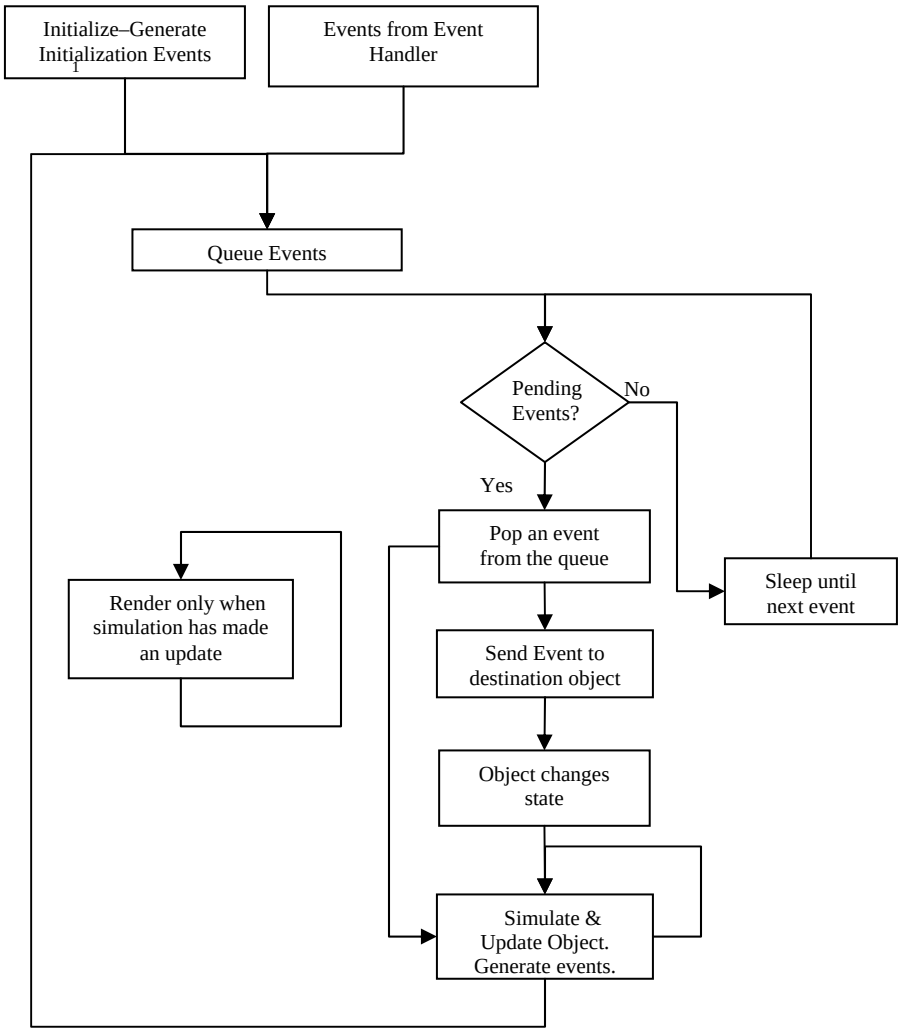

Figure 4. Discrete-Event based Simulation Engine

Resource allocation and servicing is done through the *Resource Abstraction Layer* (RAL). It is often that multiple objects will concurrently seek the same resources to perform identical or different tasks. Resource allocation and prioritization in games is especially tricky as there are real-time requirements in some resource types (i.e. graphics, networking). Simple rule-based algorithms implemented in the RAL to avoid unfair conditions will not suffice as the continuous addition of emerging types of game objects will quickly lead to rapid explosion of the number of rules in this layer. To address these resource contention issues a distributed agent algorithm was used together with *Collaborative Assignment Agents* [Seow02].

The distributed agent algorithm used is based on the Multi-Agent Assignment Algorithm (MA^3) by [Seow02a]. In the algorithm each game object possesses only localized knowledge of the resources it requires and performs BDI (Belief, Desire, Intention [Geiss99]) based reasoning before advertising resource exchange intentions. The arbitrating agent (see *Figure 5*) for the particular resources required then performs arbitrations with the intentions and assigns resources accordingly. The MA^3 algorithm is highly suitable for this application because it speeds up the lengthy arbitration process by making use of a heuristics selection process to choose near optimal heuristics, accelerating the process. Also, the algorithm allows game objects to work in parallel,

retrieving resources as collaborative resource negotiations are done using distributed decentralized agent-reasoning.
The game objects will carry out negotiations collaboratively by working out an agreement interactively that is acceptable to all agents. This is done by each game object selecting and re-selecting different resources through proposed resource exchanges that are done in the presence of an arbitration agent. *Self-Organization* is evident when each of the game objects attempts to achieve the common goal of maximizing the total number of resource exchange intentions being serviced accordingly.

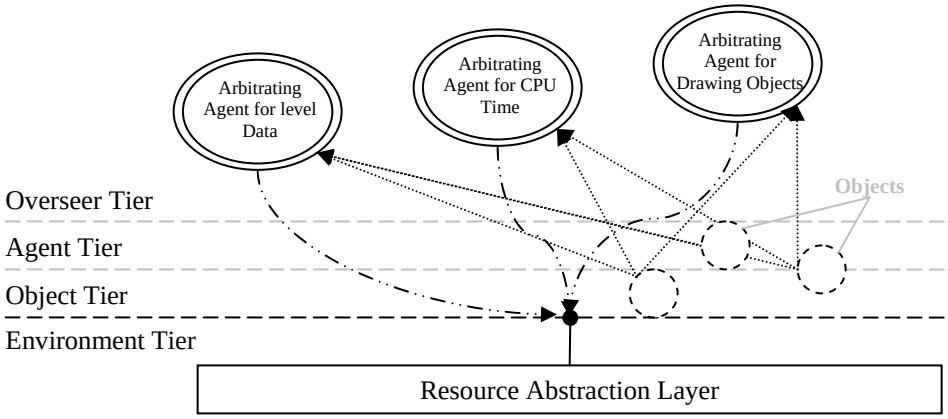

Figure 5. Arbitrating Agents performing collaborative resource assignment using MA^3

Object Tier

The Object Tier consists of the component and composite layers. This section will describe how emergent properties can be coded into game objects by taking a bottom up approach to build objects upon objects to form composite objects and eventually intelligent agent objects that exhibit emergent behavior.
It can be said that the complexity involved in implementing a game is a result of the complex structure of game objects and the interrelationships between these game objects [Wilkinson 93]. Designing game objects to have a varying number of operations and attributes to express a number of different relationships will easily increase the complexity involved in development when the MMP Game is continuously extended to include *emerging properties*. In response to that, the architecture utilizes the composite design pattern in software engineering to implement game objects so that game objects, as well as collective structures of game objects, can be treated identically.
Design patterns are proven software engineering tools that enable the successful reuse of designs and architectures. A good reference for the composite design pattern and other design patterns would be [Gamma94]. This section makes use of UML class diagrams to illustrate the organization of attributes, operations, constraints and relationships within composite objects. A good reference for UML class diagrams would be [Booch98]
Many of the game objects can be viewed as components and aggregations of components that form higher level structures or composite game objects. This aggregation of components that form complex game objects containing higher level functions and objectives are termed composite objects in object-oriented programming [Rumbaugh94]. Static interconnections and behavioral interactions of components, and global constraints

imposed on the composite objects closely resemble the workings of Complex Systems and the Complex Systems MMP Architecture in this gem [Ramazani94]. Figure 6 shows a simplified class diagram representation of how game objects were implemented as composite objects in the architecture. Composite game objects allow all the game objects to interact uniformly, through recursive composition. Figure 7 illustrates the higher level structures that form from the simpler component objects.

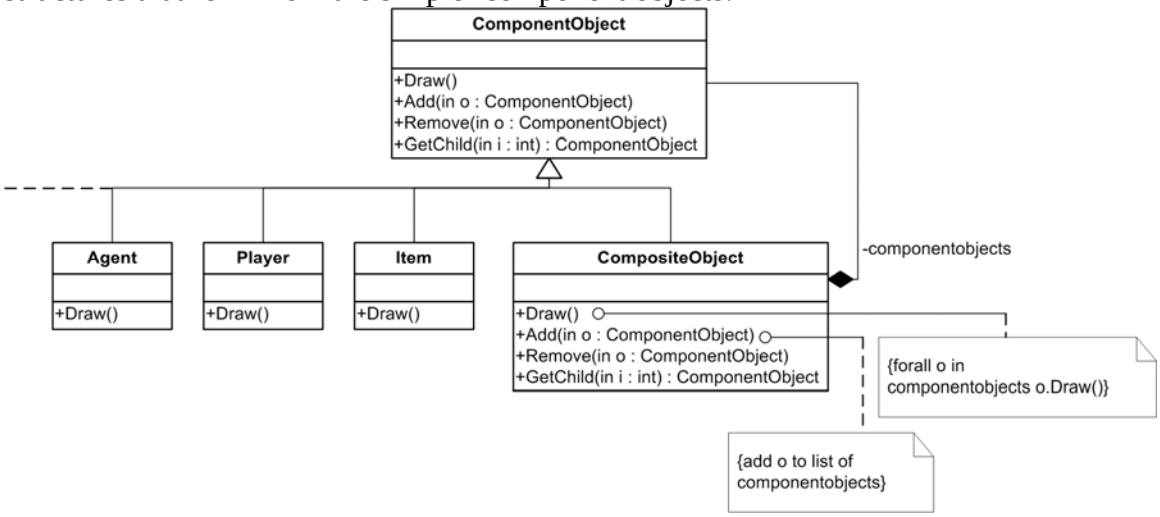

Figure 6. A Composite Object in the MMP Architecture

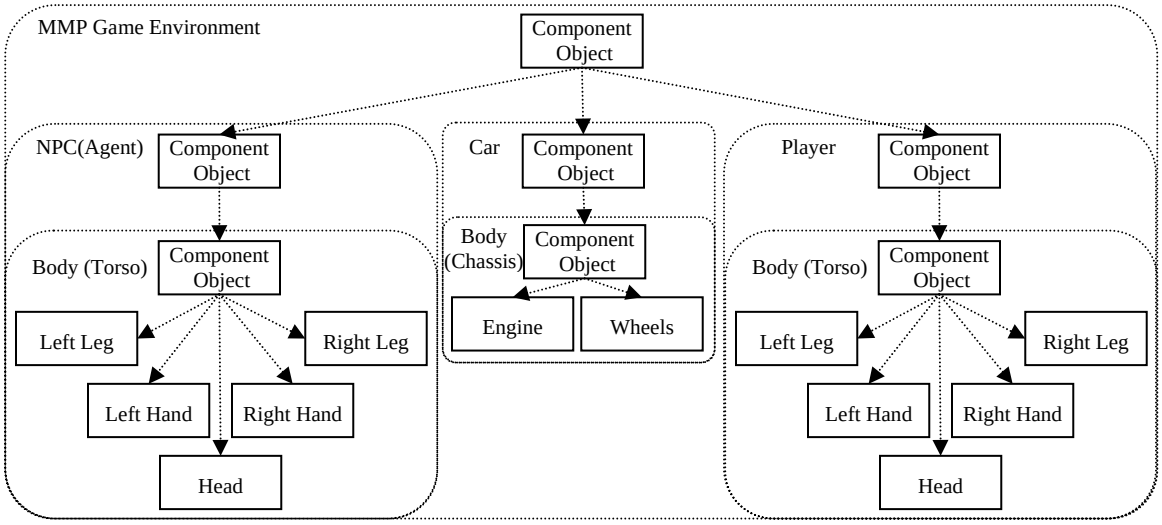

Figure 7. Higher Level structures of composite game objects from other component game objects

Composite objects offer a way to structure game objects and their semantic relationships in an organized manner. It is essential to identify how attributes, operations and relationships propagate up from simple component objects to more complex composite objects, so that effective emergence can be implemented. Attributes and operations propagate upwards in different ways. For example, when the physical weight of the object is a common attribute in all the objects, the weight of the composite object will be a sum of the weight of its component objects. However, in another example, when we consider the ambient lighting in a room with a lamp (i.e. the room being the composite

object and the lamp being the component object), the room inherits the ambient lighting conditions from the lamp, as it is. To distinguish between how attributes, operations and relationships propagate from component objects and to provide a framework to facilitate this propagation in a complex environment it is necessary to partition the attributes, operations and relationships by their inherent, aggregate and emergent properties [Ramazani94].

Inherent attributes of a game object in the architecture are attributes in the component objects of the composite game object that have logical meaning when accessed directly from the composite game object itself. From the example in figure 8, the inherent attribute of the color of the car is selected from a pool of similar inherent attributes. In the example the car game object derives its color from the chassis. To retrieve the inherent attribute the operation `GetColor()` from the `Chassis` component object of the car composite object is used as it is. This makes the `GetColor()` Operation in the car object an *inherent operation* of the composite car object as it is an operation defined by one of its component objects. The fact that only some of the components (i.e. in this case only the chassis) in the composite object participate in this relationship makes it an *inherent relationship*. Inherent relationships are common in the implementation of games as it is very simple to implement, however, it doesn't allow each of the components to contribute towards the final make-up of the higher level structure (i.e. Car). This is of little value in the MMP game architecture as the emergent properties in the higher level structures can be easily predicted.

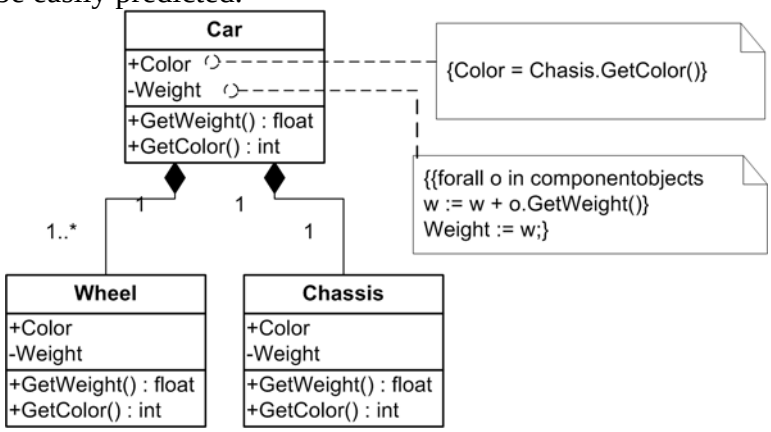

Figure 8. A composite game object car exhibiting inherent and aggregate properties

Aggregate relationships, on the other hand, are of more value to the game architecture, as the constraints in the relationship affects all component objects. In figure 8, the *aggregate attribute* Weight and the aggregate operation `GetWeight()` are used to illustrate this. A new `GetWeight()` *aggregate operation* has to be created to derive the sum of the weight attributes in the components of the composite car object. Similarly a new `Weight` aggregate attribute value is derived from this process. Changes made to the aggregate attributes in the components will require a change in the aggregate attribute of the composite object, while changes made to the aggregate attributes in the composite object might propagate down to its constituent components. It is important to understand that the new aggregation relationship *emerges* only when the component objects are associated together with a composite object (i.e. the car has only got weight when the wheels, chassis, engine etc. are attached). The aggregation relationship is highly

beneficial to the Complex Systems based MMP game architecture. However, with aggregate relationships, new behavior emerges from the aggregation of existing properties of the component objects, but new properties do not emerge in the composite objects.

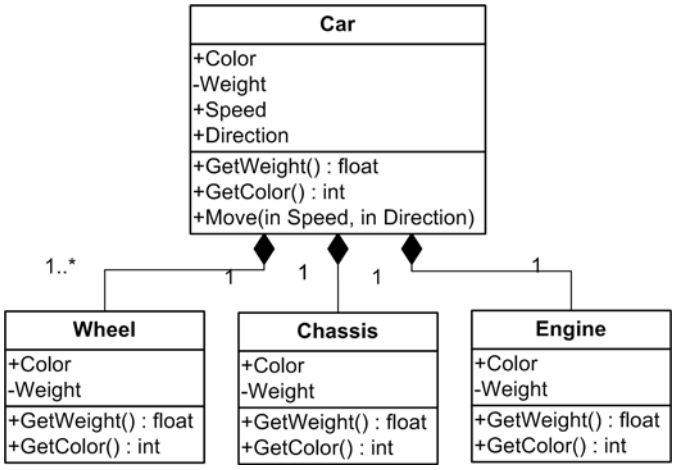

Figure 9. A composite game object car exhibiting emergent properties

In figure 9, when the Car composite object enlists a new Engine component object new *emergent attributes* of speed and direction are introduced. The `move()` operation that emerges out of this *emergent relationship* is then termed an *emergent operation*. Emergent attributes are unique as they identify specific attributes that characterizes the composite objects as a whole and not as just a combination of its parts. The emergent operation on the other hand is unique because it introduces a new operation that is not reliant on the component objects. (i.e. The car is capable of movement, as long as the engine, wheels, chassis, etc. are present)

Emergent relationships make the implementation of a Complex Systems based MMP architecture feasible as it supports the bottom-up creation of game objects. Emergent relationships enable the encapsulation of the components of the composite objects, as client objects only interact with the composite object that exhibit emergent properties. Emergent relationships reduce the complexity involved in development as it enables developers to treat composite objects as if they were individual objects.

However, the challenge in this architecture is to maintain the functional validity of the emergent properties during game play. This is done by ensuring that the functional meaning of the emergent relationships between component and composite objects are valid. Like in figure 10, this can be easily implemented as a table of emergent relationships. For example, if the engine of a car is damaged after knocking into a wall, the particular relationship can be marked as invalid and access to the emergent properties of `Speed`, `Direction` and `Move()` can be denied. The observer design pattern can then be used in conjunction with the Emergent Relationship table to implement an Observer object for each component. The Observer object maintains a one-to-many dependency between objects so that changes in the component observed will be communicated to the appropriate composite objects [Gamma94].

Emergent Relationships

Composite Object	Emergent Relationship	Component Object	Emergent Properties
Car	Movement	Wheel	Speed, Direction, Move()
Car	Movement	Engine	Speed, Direction, Move()
⋮	⋮	⋮	⋮

Figure 10. The Emergent Relationship Table

Agent Tier

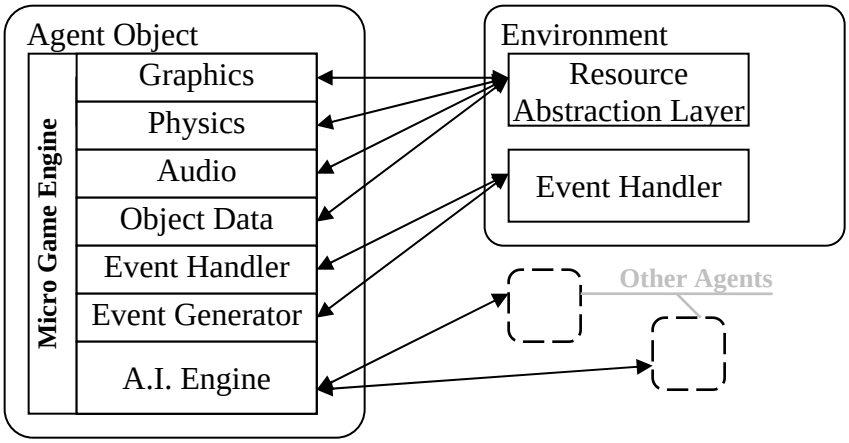

Figure 11. Components of an Agent

The Agent Tier hosts the NPCs, human players and other game objects that exhibit a significant-level of artificial intelligence in the MMP architecture. Each agent object embodies within itself a micro game engine (see *Figure 11*) that contains attributes of the agent and agent-specific routines of the game engine. Conceptually, each agent object stores its own game data within itself. The actual implementation differs, as game data and other resources are pooled together and stored in a central location to facilitate efficient memory management. Initially, agents negotiate for resources through the arbitrating agents. After resources are allocated the agents communicate directly with the resource abstraction layer to access allocated resources. Resources include processes and threads to run custom game loops and operations within each agent object. An Event Handler in the agent object processes events routed to it by the Event Handler in the Environment Tier. Agents can create events to perform updates and other operations through the event generator. This will prompt the Simulation Engine or other engines to act accordingly.

The A.I. Engine communicates with other engines to affect its decision making process (see *Feedback based decision making system*). The Agent Tier was designed to be platform independent so that high-level concepts found in Complex Systems can be easily extended upon to improve the MMP game experience. The next section covers the functions of the last tier which is the Overseer Tier.

Overseer – Tier

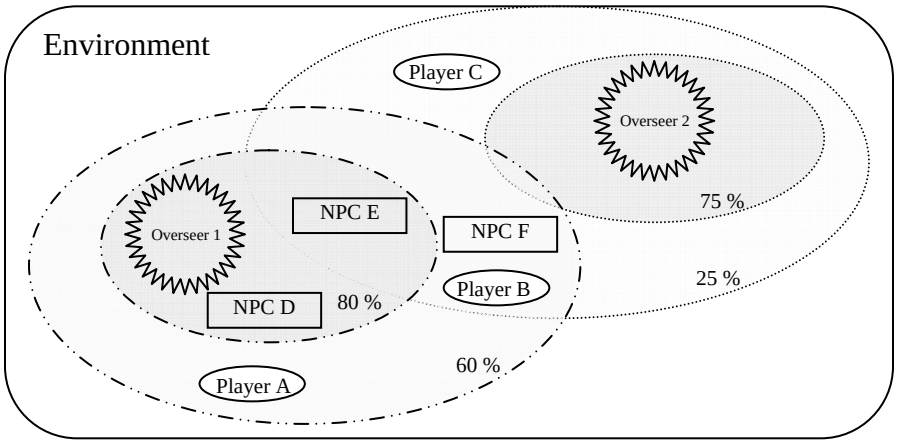

Figure 12. Overseers influencing agent-behavior via Policy based control

The decentralized approach to control adopted by most Complex Systems is also present in this architecture. Agents with the means to influence the actions of other agents directly were introduced to exercise policy based control of game objects in the virtual game environment. These agents have been termed *overseers*. The overseers play a major role in administering policies so that emergent properties in the game environment that can possibly cause the system to behave absurdly are removed. However, great care must be taken to ensure that the policies introduced are not overly dominant so as to not suppress emergence.

Multiple overseers allow different policies from different policy-makers to affect a different niche-market of players. The segregation and grouping of players can be done in many ways. Grouping can be done by cultural diversity to allow overseers to ensure that cultural values are represented faithfully. It can be done by age groups to allow relevant content to not reach minors. To allow multiple individual policy makers to influence the behavior of agents and to sustain the many-to-many relationship in the complex environment, overseers are used to influence the behavior of agents to varying degrees.

In figure 12, the overseers' influence on other game objects decreases with increasing spatial differences. Agent E is largely affected by Overseer 1's game policies and partially affected by Overseer 2's game policies. Agent F is partially affected by both Overseers policies. Agent D is only affected by Overseer 2's policies. The spatial partitioning can occur in a number of different ways.

For example, Overseer 1 can be used to govern Army A in an MMORTS while Army B can be governed by Overseer 2. Both Overseers will then implement policies that are common to both armies. When a battle occurs between both armies, a spatial partitioning can be made with the length of service of each soldier (i.e. agent), to determine the probability of defection of each soldier. As the physical distance between the Overseers becomes increasingly smaller as the armies approach each other, each of the Overseers will then attempt to force agents in the opponent teams to defect.

In another example that makes use of figure 12, if we make use of both Overseers to implement in-game economic policies, we might observe that player B's economic model is governed more by Overseer 1 than Overseer 2 as Player B falls in between an

influence level of 60-80% from Overseer 1, in contrast to the 25-75% influence level of Overseer 2.
Other game rules have also been implemented as overseer policies. These include policies to encourage fair play, prevent collusion between players, track abnormal behavior and employ cheat detection.

Feedback based decision making system

The *feedback based decision making system* is part of the AI Engine in the agents and overseers. Most game designs have in place a feedback system to allow agents in the game to make decisions based on only the current game state at the time the decision is made. This architecture is rigid and supports little evolving characteristics for the game objects, as they are only influenced by non-evolving game rules. There have been many efforts to break this rule-based paradigm. Machine learning [Alexander02] and neural networks [Manslow02] have been used to provide some level of randomness in the system. The feedback model used in this architecture, when used in unison with these techniques improves the emergent behavior of the system. In the feedback based agent-agent interaction model presented in figure 13, agent behavior is no longer influenced by just its AI Engine and feedback from its actions, but is instead influenced by the actions of other agents, who are in turn influenced by other agents, thus introducing *cross term inducing features* [LeBlanc00] into the environment.
The game state holds the state of all attributes in the virtual world. When the agents make decisions they require the current state of the world, the external inputs that triggered the decision making process and the responses to similar actions taken by other agents. When heterogeneous agents with varying decision making processes interact with each other the unpredictability of decisions made increases.
It is assumed that human players learn only from the behaviors of other agents in addition to the game responses to their actions just like the other agents in the architecture. However, this is not entirely true as human players also take into account their real-world experiences before making a decision. This allows a form of natural randomness to emerge in this virtual game environment as human responses are also fed back into the system. Overseers are also part of the feedback model. Their main role is to allow only desirable agent behavior to propagate among other agents over time. There are a number of methods to implement the feedback based decision making model in figure 13. The one used in the MMP architecture is quite complex and is beyond the scope of this gem. [Norlig00] and [Rogova03] offer insight into these particularly complex decision making processes. [Evans02] presents a much simpler architecture that makes use of BDI Reasoning [Geiss99] together with a perceptron model that make use of the weighted average of the feedback and inputs received for effective decision making.

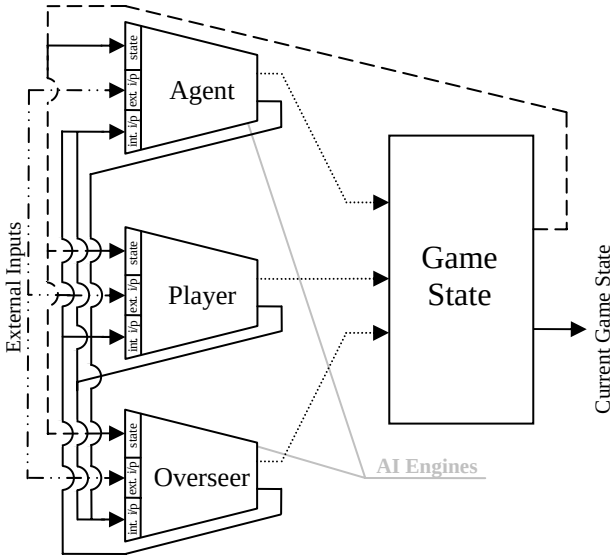

Figure 13. Overseers influencing agent-behavior via Policy based control

Conclusion

Developing Massively Multiplayer Games that can present players with new interesting challenges, behavior and content is not a trivial task. The architecture presented in this gem can be used as a foundation to develop Massively Multiplayer games that exhibit emergent properties and behavior that is much needed by next generation Massively Multiplayer Games.

References

[Kirschbaum98] Kirschbaum, David, "Introduction to Complex Systems," available online at http://www.calresco.org/intro.htm, October 1998

[Odell02] Odell, James, "Agents & Complex Systems," Journal of Object Technology Vol. 1, No. 2:pp. 35-45, available online at http://www.jot.fm/issues/issue_2002_07/column3 , July 2002.

[Wikipedia05] Wikipedia, "Complex Systems," available online at http://en.wikipedia.org/wiki/Complex_system, July 20, 2005.

[Ageia05] Ageia, "A White Paper : Physics, Gameplay and the Physics Processing Unit," available online at http://www.ageia.com/pdf/wp_2005_3_physics_gameplay.pdf March, 2005.

[Garcia04] Garcia, Inmaculada, Molla, Ramon & Barella, Toni, "GDESK: Game Discrete Event Simulation Kernel", *Proceedings of the 12th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* 2004 (WSCG 2004) available online at http://wscg.zcu.cz/wscg2004/Papers_2004_Full/E67.pdf .

[Seow02] Seow, Kiam Tian & How, Khee Yin, "Collaborative assignment : a multiagent negotiation approach using BDI concepts," Proceedings of the first international conference on Autonomous agents and multiagent systems: part 1 (ICAA 2002):pp.256-263.

[Seow02a] Seow, Kiam Tian & Wong, Kok Wai, “Collaborative Assignment: Using Arbitrated Self-Optimal Initializations for Faster Negotiation,” 2002.
[Evans02] Evans, Richard, *Varieties of Learning, AI Programming Wisdom*, Charles River Media 2002.
[Geiss99] Geiss,W., *Multiagent System : A Modern Approach to Distributed Artificial Intelligence*, The MIT Press, 1999.
[Wilkinson93] Wilkinson, M., Byers, P., "The engineering of complex systems," IEE Computing & Control Engineering Journal (August 1993): pp. 187-189
[Gamma94] Gamma, Erich et al., *Design Patterns*, Addison Wesley Longman, Inc., 1994.
[Booch98] Booch, Grady, *The Unified Modeling Language User Guide*, Addison Wesley, 1998.
[Rumbaugh94] Rumbaugh, J., "Building boxes: Composite Objects," JOOP Vol.7, No. 7 (November 1994): pp. 12-22
[Ramazani94] Ramazani, Dunia, "Contribution of Object-Oriented Methodologies to the Specification of Complex Systems," IEEE Engineering of Complex Computer Systems, (ECCS 1995): pp. 183-186
[Alexander02] Alexander, Thor, *GoCap: Game Observation Capture, AI Programming Wisdom*, Charles River Media 2002.
[Manslow02] Manslow, John, *Imitating Random Variables in Behavior Using a Neural Network, AI Programming Wisdom*, Charles River Media, 2002.
[LeBlanc00] LeBlanc, Marc, “Formal Design Tools - Emergent Complexity & Emergent Narrative,” Proceedings of the Game Developer’s Conference (GDC 2000).
[Rogova03] Rogova, G., Lollett, C. & Scott, P., “Utility-Based Sequential Decision-Making In Evidential coorperative Multi-Agent Systems,” Proceedings. *Proceedings of the Sixth International Conference of Information Fusion* 2003.
[Norlig00] Norling, E., Sonenberg, L. & Ronnquitst, R., “Enhancing Multi-Agent Based Simulation with Human-Like Decision Making Strategies”, *Proceedings of Second International Workshop on Multi-Agent-Based Simulation* 2000 Boston, MA, USA, available online at http://citeseer.ist.psu.edu/norling00enhancing.html