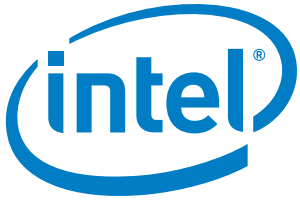


Designing the Framework of a Parallel Game Engine

How to Optimize Your Game Engine for Intel® Multi-Core CPUs

By Jeff Andrews

White Paper

Visual Computing

Multi-Threading

Abstract Summary

With the advent of multiple cores within a processor, the need to create a parallel game engine has become increasingly important. Although it is still possible to focus primarily on only the GPU and have a single-threaded game engine, the advantages of using all the processors on a system, whether CPU or GPU, can give the user a much greater experience. For example, by using more CPU cores, a game can increase the number of rigid body physics objects for greater effects on the screen. Optimized games might also yield a smarter AI.

This white paper covers the basic methods for creating and optimizing a multi-core game engine. It also describes the state manager and messaging mechanism that keeps data in sync and offers several tips for optimizing parallel execution. Multiple block diagrams are used to depict theoretical overviews for managing tasks and states, interfacing with scenes, objects, and tasks, and then initializing, loading, and looping within synchronized threads.

Table of Contents

Introduction

The Parallel Game Engine Framework or *engine* is a multi-threaded game engine that is designed to scale to all available processors within a platform. To do this, the engine executes different functional blocks in parallel so that it can use all available processors. However, this is often easier said than done, because many pieces in a game engine often interact with one another and can cause many threading errors. The engine takes these scenarios into account and has mechanisms for getting the proper synchronization of data without being bound by synchronization locks. The engine also has a method for executing data synchronization in parallel to keep serial execution time to a minimum.

Parallel Execution State

The concept of a parallel execution state in an engine is crucial to an efficient multi-threaded runtime. For a game engine to truly run in parallel, with as little synchronization overhead as possible, each system must operate within its own execution state and interact minimally with anything else going on in the engine. Data still needs to be shared, but now instead of each system accessing a common data location to get position or orientation data, for example, each system has its own copy—removing the data dependency that exists between different parts of the engine. If a system makes any changes to the shared data, notices are sent to a state manager, which in turn queues the changes, called *messaging*. Once the different systems have finished executing, they are notified of the state changes and update their internal data structures, which is also part of messaging. Using this mechanism greatly reduces synchronization overhead, allowing systems to act more independently.

Execution Modes

Execution state management works best when operations are synchronized to a clock, which means the different systems execute synchronously. The clock frequency may or may not be equivalent to a frame time, and it is not necessary for it to be so. The clock time does not even have to be fixed to a specific frequency, but instead can be tied to frame count, such that one clock step is equal to how long it takes to complete one frame, regardless of length. The implementation one chooses to use for the execution state will determine the clock time. Figure 1 shows the different systems operating in the *free step mode* of execution, which means they don't have to complete their execution on the same clock. There is also a *lock step mode* of execution (Figure 2) in which all systems complete in one clock. The main difference between the two modes is that free step provides flexibility in exchange for simplicity, while lock step is the reverse.

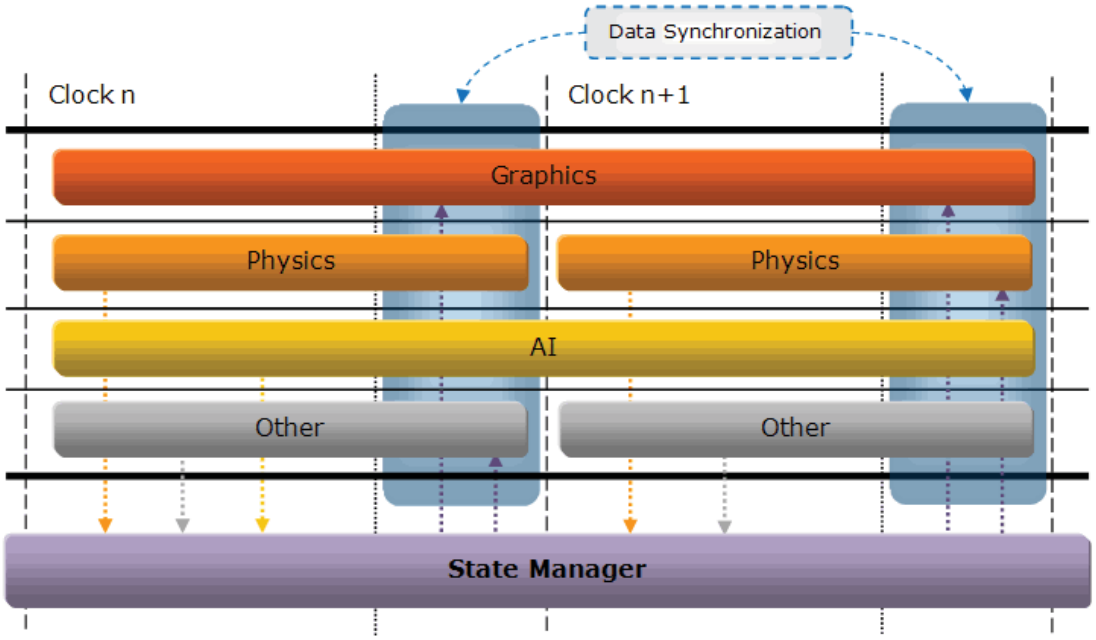


Figure 1. Execution state using the free step mode.

Free Step Mode

This execution mode allows a system to operate in the time it needs to complete its calculations. "Free" can be misleading, because a system is not free to complete whenever it wants, but is free to select the number of clocks it needs to execute.

With this method, a simple notification of a state change to the state manager is not enough. Data must also be passed along with the state-change notification because a system that has modified shared data may still be executing when another system that wants the data is ready to do an update. This requires the use of more memory and copies, so it may not be the most ideal mode for all situations. Given the extra memory operations, free step might be slower than lock step, although this is not necessarily true.

Lock Step Mode

With this mode all systems must complete their execution in a single clock. Lock step mode is simpler to implement and does not require data to be passed with the notification because systems that are interested in a change made by another system can simply query that system for the value (at the end of execution).

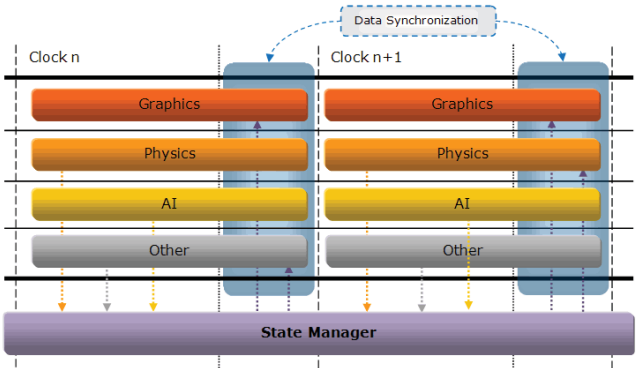


Figure 2. Execution state using the lock step mode.

The lock step mode can also implement a pseudo-free-step mode of operation by staggering calculations across multiple steps. One use for this might be with an AI that calculates its initial "large view" goal in the first clock, then instead of just repeating the goal calculation for the next clock comes up with a more focused goal based on the initial goal.

Data Synchronization

It is possible for multiple systems to make changes to the same shared data. To do this, the messaging needs some sort of mechanism that can determine the correct value to use. Two such mechanisms can be used:

- Time. The last system to make the change time-wise has the correct value.
- Priority. A system with a higher priority is the one that has the correct value. This can also be combined with the time mechanism to resolve changes from systems of equal priority.

Data values that are determined to be stale, via either mechanism, will simply be overwritten or thrown out of the change notification queue.

Using relative values for the shared data can be difficult because some data may be order-dependent. To alleviate this problem, use absolute data values so that when systems update their local values they replace the old with the new. A combination of both absolute and relative data is ideal, although its use depends on each specific situation. For example, common data, such as position and orientation, should be kept absolute because creating a transformation matrix for it depends on the order in which the data are received. A custom system that generated particles, via the graphics system, and fully owned the particle information could merely send relative value updates.

The Engine

The engine's design focuses on flexibility, allowing for the simple expansion of its functionality. It can be easily modified to accommodate platforms that are constrained by certain factors, such as memory.

The engine is broken up into two distinct pieces: the *framework and the managers*. The framework contains the parts of the game that are duplicated; that is, there will be multiple instances of them. It also contains items that have to do with execution of the main game loop. The managers are singletons that the game logic depends on.

Figure 3 illustrates the different sections that make up the engine.

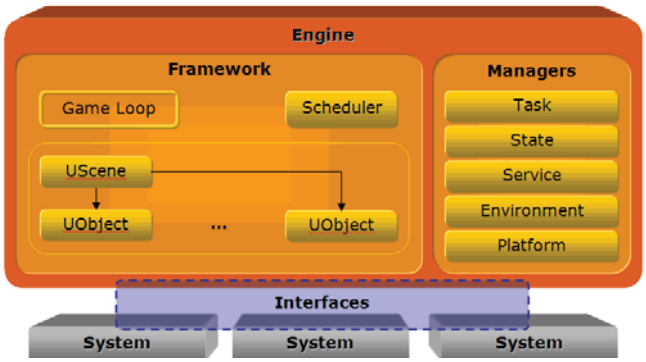


Figure 3. The high-level architecture of the engine.

Notice that the game processing functionality, referred to as a *system*, is treated as a separate entity from the engine. This modularity makes the engine the "glue" for tying all the functionality together. Modularity also allows the systems to be loaded or unloaded as needed.

The interfaces are the means of communication between the engine and the systems: The systems implement the interface so that the engine can access a system's functionality, and the engine implements the interface so that the systems can access the managers.

For more information about this concept, refer to Appendix A.

As described in the Parallel Execution State section, the systems are inherently discrete—systems can run in parallel without interfering with the execution of other systems. This does cause some problems when systems need to communicate with each other, because the data is not guaranteed to be in a stable state. Two reasons for inter-system communication are:

- To inform another system of a change made to shared data, such as position or orientation.
- To request functionality that is not available within a particular system, such as the AI system asking the geometry or physics system to perform a ray intersection test.

The first communication problem is solved by implementing the state manager mentioned in the previous section.

To solve the second problem, a mechanism is included through which a system can provide a service that a different system can use.

Framework

The framework ties all the different pieces of the engine together. Engine initialization occurs within the framework, with the exception of the managers, which are globally instantiated. The information about the scene is also stored in the framework. For flexibility, the scene is implemented as what is called a *universal scene*, which contains *universal objects*—containers for tying together the different functional parts of a scene.

The game loop is also located within the framework. Figure 4 shows its flow.

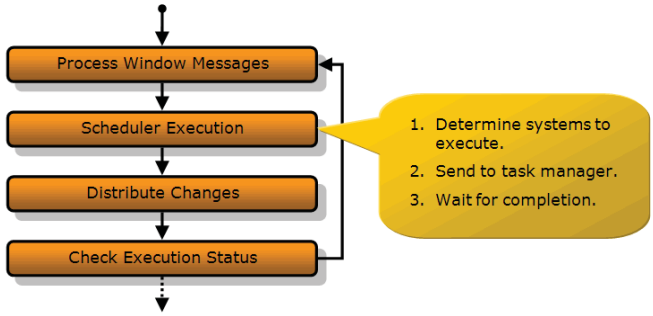


Figure 4. The main game loop.

The first step in the game loop is to process all pending OS window messages because the engine operates in a windowed environment. The engine will be unresponsive to the OS if this is not done. The scheduler next issues the systems' tasks with the task manager, and then the changes that the state manager has been keeping track of are distributed to all interested parties. Finally, the framework checks the execution status to see if the engine should quit or perform some other engine execution action, such as go to the next scene. The engine execution status is located in the environment manager, which is described later in this article.

Scheduler

The scheduler holds the master clock for execution, which is set at a pre-determined frequency. The clock can also run at an unlimited rate, for things like benchmarking mode, so that there is no waiting for the clock time to expire before proceeding.

The scheduler submits systems for execution, via the task manager, on a clock tick. For free step mode, the scheduler communicates with the systems to determine how many clock ticks they will need to complete their execution. Then it determines which systems are ready for execution and which ones will be finished by a certain clock tick. The scheduler can adjust this amount if it determines that a system

needs more execution time. Lock step mode has all systems start and end on the same clock, so the scheduler will wait for all systems to complete execution.

Universal Scene and Objects

The universal scene and objects are containers for the functionality that is implemented within the systems. By themselves, the universal scene and objects have no functionality other than being able to interact with the engine. They can, however, be *extended* to include the functionality that is available in a system, which gives them the ability to take on the properties of any available system without having to be tied to a specific system. This is called *loose coupling*. Loose coupling is important because it allows the systems to be independent of each other and run in parallel.

Figure 5 illustrates the universal scene and object extension of a system.

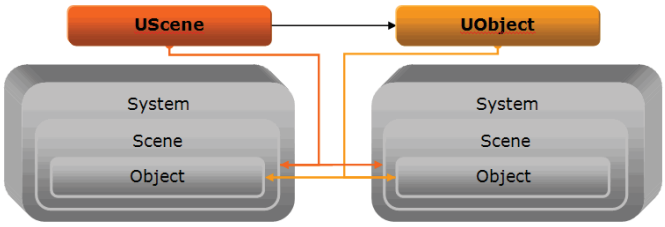


Figure 5. The universal scene and object extension.

Here's how extensions work: A universal scene is extended to have graphics, physics, and other properties. The graphics scene extension initializes the display and other things, and the physics scene extension sets up the rigid body world, such as gravity. Scenes contain objects, so a universal scene has several universal objects. Similarly, the universal object, contained in the universal scene, is extended to have graphics, physics, and other properties. The graphics object extension draws the object on screen, and the physics object extension is responsible for the rigid body interaction of the object with other rigid bodies.

For a more detailed diagram showing the relationship between the engine and the systems, refer to Appendix B.

The universal scene and universal objects are also responsible for registering their extensions with the state manager, so that the extensions are notified of changes made by other extensions (that is, other systems). For example, the graphics extension would be registered to receive notification of position and orientation changes made by the physics extension.

The Managers

The managers provide global functionality within the engine and are implemented as singletons; that is, only one instantiation is made available for each type of manager. Managers are singletons because duplication of their resources can cause redundancy, leading to potential processing performance implications. Singletons provide common functionality that is usable across all the systems.

Task Manager

The task manager handles the scheduling of a system's task within its thread pool. The thread pool creates one thread per processor to get the best possible n-way scaling to processors. The task manager also prevents over-subscription, which avoid unnecessary task switching within the OS.

The task manager receives from the scheduler its list of tasks to execute, as well as which tasks to wait for execution to complete. The scheduler gets its list of tasks to execute from the different systems themselves. Each system has only one primary task; this is called *functional decomposition*. Each primary task can issue as many sub-tasks as it wants for operating on its data; this is called *data decomposition*.

Figure 6 shows how the task manager could issue tasks onto threads for execution on a quad-core (four-thread) system:

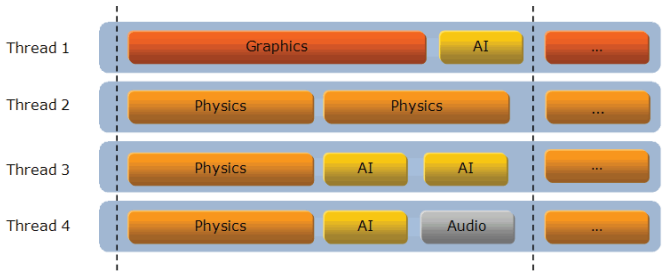


Figure 6. An example of the task manager's four-thread pool.

Aside from access by the scheduler for issuing primary tasks, the task manager also has an initialization mode in which it calls systems serially from each thread so that the systems can initialize any local storage they require for execution.

For help getting started on implementing a task manager, refer to Appendix D.

State Manager

State management is part of the messaging mechanism that tracks and distributes change notifications made by a system to other interested systems. To reduce unnecessary change notification broadcasts, systems must register with the state manager for the changes they want to receive. This mechanism is based on the observer design pattern, which is described in more detail in Appendix C. The basic premise of the observer design pattern is an *observer* observing a *subject* for any changes, with a *change controller* acting as a mediator between the two.

Here's how the mechanism works:

1. The observer registers the subject it wants to observe with the change controller (or state manager).
2. When the subject has changed one of its properties, it sends a change notification to the change controller.
3. The change controller, when requested by the framework, distributes the subject's change notification to the observer.
4. The observer queries the subject for the actual changed data.

The free step mode of operation introduces extra complexities into this mechanism: (1) The data must be included with the change notification because a system that has modified shared data may still be executing and therefore cannot be queried for its value. (2) If a system is not yet ready to receive the changes at the end of a clock tick, the state manager will need to hold onto that data until all systems registered for it are finally ready to receive it.

The framework implements two state managers—one for handling changes on the scene level and another for handling changes on the object level. Because the scenes and objects, for the most part, have different messages that are relevant to them, separating them removes the need to process unnecessary messages. However, any object changes that are relevant to the scene will be registered with the scene so that it will receive those change notifications.

To remove any synchronization overhead, the state manager has a change queue for each thread created by the task manager. No synchronization is required when accessing the queue. The queues can then be merged after execution using the method described in the Data Synchronization section.

Although it seems that change notifications would have to be distributed serially, this action can be parallelized. When systems are executing their tasks they operate across all their objects. For example, the physics system is moving around objects, checking for collisions, and setting new forces as physics objects interact with each other. During change notification, a system's object is no longer interacting with other objects from its own system, but is now interacting with other extensions in the universal object it is associated with. This means that universal objects are now independent of each other, so each universal object can be updated in parallel. Take note, though, that there may be some corner cases that need to be accounted for with synchronization. Still, something that once looked hopelessly serial can now get at least some parallelization.

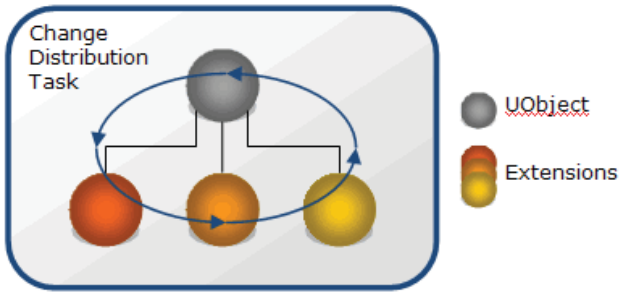


Figure 7. An internal UObject change notification.

Service Manager

The service manager provides access to functionality to systems that are external to their implementations. The service manager does not provide this directly but has the interfaces defined for it, and any systems that implement the exposed interface functionality will register themselves with the service manager.

Only a small set of services is available because the design of the engine is to keep the systems running as discretely as possible. Also, the systems are not free to provide any service they so choose, but only the ones that the service manager provides.

The service manager also gives the different systems access to each other's properties. Properties are values of each system that are specific to a system and are therefore not passed in the messaging system. Examples: the screen resolution of the graphics system or the gravity value of the physics system. The service manager gives the systems access to these properties without giving the systems direct control over them. The property changes are queued and are issued only during serial execution. Accessing another system's properties is a rare occurrence and should not be used as common practice. This is made available for properties or functionality that does not change from frame to frame. Examples: the console window turning on and off the wireframe mode in the graphics system or the user interface system changing the screen resolution as requested by the user.

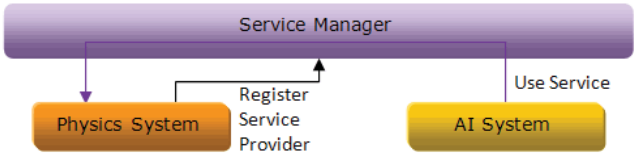


Figure 8. An example of the service manager.

Environment Manager

The environment manager provides the functionality for the engine's running environment. The function groups provided by the environment manager are:

- **Variables.** The variable names and data that are shared across the entire engine. The variables are usually set upon loading a scene or user settings, and are queried in the engine and by the different systems.
- **Execution.** Information about the execution, such as the end of a scene or the program. This can be set or queried by either the engine or the systems.

Platform Manager

The platform manager handles all OS call abstraction and also provides added functionality beyond just a simple abstraction. This gives the benefit of encapsulating several common functional steps within one call, instead of all the callers having to implement them or know about the nuances of the OS calls.

An example is the call in the platform manager to load a system's dynamic library. In addition to loading a system in, the platform manager also gets the function entry points and then calls the library's initialization function. It also keeps around a handle to the library, unloading it upon exiting the engine.

The platform manager provides information about the processor, such as which instructions sets are supported, frequency, and cache sizes. The platform manager also initializes some of the behavior for the executing process, such as the operation of floating-point denormals (or subnormal numbers).

Interfaces

The interfaces are the way the framework, the managers, and the systems communicate with each other. The framework and the managers reside within the engine, and therefore the framework has direct access to the managers. The systems, however, reside outside of the engine and have different functionality from one another, making it necessary to have a common method for accessing them. Also, the systems do not have direct access to the managers, so the systems also need a method for accessing the managers. Systems do not require access to the full functionality of the managers, as certain items should only be accessible to the framework.

The interfaces provide a set functionality that needs to be implemented in order to have a common method of access. This makes it unnecessary for the framework to know the details about a specific system as it can communicate to it through a known set of calls.

Subject and Observer

The subject and observer interfaces register the observer with the subject and pass the change notifications from the subject to the observer. A default subject implementation is also provided because the functionality to handle observer registration/de-registration is common to all subjects.

Manager

The managers, even though they are singletons, are only directly available to the framework; the different systems cannot access them. In order to provide access, each manager would have an interface that exposes a subset of its functionality. The interface would then be passed to the system when it gets initialized and the systems would then have access to a subset of the manager.

The interface defined is dependent upon the manager and therefore is not a common interface but specific to that manager.

System

The systems need to implement interfaces in order for the framework to get access to its components. Without system interfaces, the framework would have to implement a specific implementation of each new system that gets added to the engine.

A system has four components; in turn, a system must implement four interfaces.

- System interface. Provides methods for creating and destroying scenes.
- Scene interfaces. Provide methods for creating and destroying objects and a method for retrieving the primary task.
- Object interfaces. Typically associated with what is visible on-screen to the user.
- Task interface. Used by the task manager when issuing tasks within its thread pool.

The scene and object interfaces also derive from the subject and observer interfaces because these are the pieces of the system that need to communicate not only with one another, but also with the universal scene and objects to which they are attached.

Change

Change interfaces are special interfaces that are used for passing data between the systems. Any systems that make these specific modifications must also implement the interface. For example, the geometry interface has methods for retrieving the position, orientation, and scale for a certain item. Any systems that make modifications to geometry would need to implement this interface so that a different system would be able to access the geometry changes without needing to know about the other system.

Systems

The systems provide the game functionality to the engine. Without them, the engine would just spin endlessly without any tasks to perform. To keep the engine from having to know about the different system types, systems must implement the interfaces described in the System section. This makes it simpler to add a new system to the engine because the engine doesn't need to know about the details.

Types

The engine should have some predefined systems types for standard game components. Examples: geometry, graphics, physics (rigid body collision), audio, input, AI, and animation.

A custom type is also recommended for systems that implement functionality outside the common functional blocks in a game. Any systems that modify the custom type's specific data items will need to know about the custom type's interface because the engine does not provide this information.

System Components

A system has several components that need to be implemented: system, scene, object, and task. These components are all used to communicate with the different sections within the engine (Figure 9).

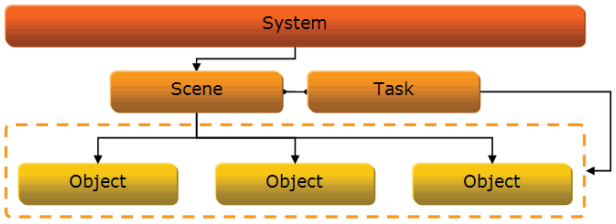


Figure 9. The relationship between the system components.

For a more detailed diagram of the systems' relationship, refer to Appendix A.

System

The system component, or *system*, initializes the system resources that will remain more or less constant throughout the engine's execution. For example, the graphics system analyzes the passed-in resource locations to determine where they are located for quicker loading upon use of a resource. The screen resolution is another item set by the graphics system.

The system is also the main entry point for the framework and provides information about itself, such as its type, and methods for scene creation and destruction.

Scene

The scene component, also known as a *system scene*, handles the resources that are pertinent to the existing scene. The universal scene uses this scene as an extension of its functionality to make available the properties this system scene provides. An example of this component is the physics scene creating a new world and setting the gravity for the world upon scene initialization.

The scene provides methods for object creation and destruction. It also owns the task component, which is used to operate on the scene.

Object

The object component, also called a system object, is an object within the scene and is typically associated with what is visible to the user on-screen.

The universal object uses the system object as an extension of its functionality, described in "Universal Scene and Objects", to allow the properties this object provides to be exposed via the universal object. For example, a universal object could extend geometry, graphics, and physics to create a beam of wood on-screen. The geometry system would hold the position, orientation, and scale information of the object. The graphics system would display the beam on-screen using the given mesh, and the physics system would apply rigid body collision to the beam so that it would correctly interact with other objects and with gravity.

In certain situations a system object may be interested in the changes of a different universal object, or one of the universal object's extensions. In this case a *link* can be established so that the system object can observe the other object.

Task

The task component, referred to as a system task, operates on the scene. When the task receives a command to update from the task manager, the task performs the system's functionality on the objects within the scene.

The task can also choose to subdivide its execution into subtasks and schedule the subtasks with the task manager for even more threaded execution. Doing this allows the engine to scale more readily to a configuration with multiple processors. This technique is known as *data decomposition*.

During the task's update of the scene, any modifications done to its objects are posted to the state manager.

Tying It All Together

The engine execution can be broken up into several stages: initialization, scene loading, and game loop .

Initialization

Engine execution begins by initializing the managers and the framework.

1. The framework calls the scene loader to load in the scene.
2. The loader determines what systems the scene is using, and then calls the platform manager to load those modules.
3. The platform manager loads the modules, passes in the manager interfaces, and then calls into them to create a new system.
4. The system module returns a pointer to the instantiated system that implements the system interface.
5. The system module registers any services it provides with the service manager.

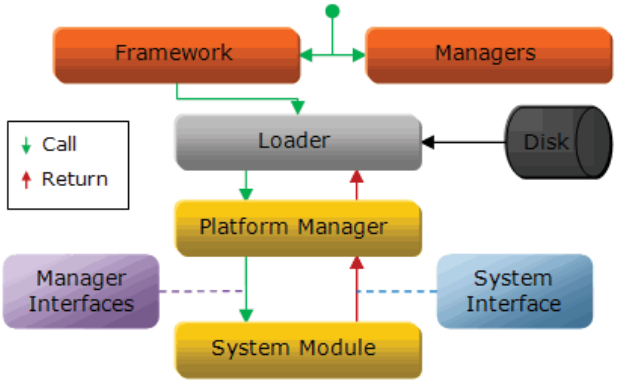


Figure 10. The engine manager and system initializations.

Scene-Loading

Control returns to the loader, which loads the scene.

1. The loader creates a universal scene and calls each system interface to instantiate system scenes, extending the functionality of the universal scene.
2. The universal scene checks each system scene for any shared data changes they could make and for shared data changes they want to receive.
3. The universal scene then registers the matching system scenes with the state manager so they will be notified of the changes.
4. The loader creates a universal object for each object in the scene and determines which systems will be extending the universal object. The universal object follows a similar system object registration pattern with the state manager, as occurs for the universal scene.
5. The loader instantiates system objects via the system scene interfaces it previously received and extends the universal objects with the system objects.
6. The scheduler then queries the system scene interfaces for their primary tasks because the scheduler is responsible for issuing the primary tasks to the task manager during execution.

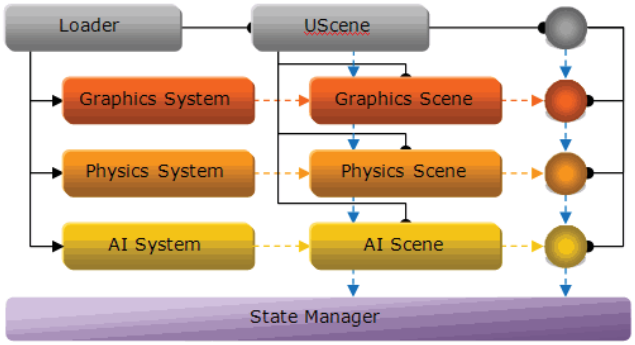


Figure 11. The universal scene and object initializations.

Game Loop

The main game loop begins processing.

1. The platform manager is called to process all window messages and other platform-specific items that are needed for operation on the current platform.
2. Execution is then transferred to the scheduler, which waits for the clock time to expire before proceeding.
3. The scheduler, for free step mode, checks which of the system tasks completed execution in the previous clock. All tasks that have finished (that is, they are ready to execute) get issued to the task manager.
4. The scheduler now determines which tasks will complete on the current clock and waits for completion of those tasks.
5. For lock step mode, the scheduler issues all tasks and waits for them to complete for each clock step.

Task Execution

Execution is transferred to the task manager.

1. The task manager queues all submitted tasks and starts processing each one as threads become available. (Task processing is specific to each system. Systems can operate using only one task or they can issue more tasks which get queued in the task manager, thus potentially getting executed in parallel).
2. As tasks execute they operate on the entire scene or on specific objects and modify their internal data structures.
3. Any data that is considered shared, such as position and orientation, need to be propagated to the other systems. To do this, the system task has the system scene or system object (whichever was changed) inform their observer of the change. In this case the observer is actually the change controller located in the state manager.
4. The change controller queues the change information to be processed later, but change types that the observer is not interested in are simply ignored.
5. If the task needs any services it goes through the service manager to call into the provided service. The service manager can also be used to change the property of a different system that isn't exposed via the messaging mechanism (that is, the user input system changes the screen resolution of the graphics system).
6. Tasks can also call into the environment manager to read environment variables, and change the runtime state (pause execution, go to next scene, and so on).

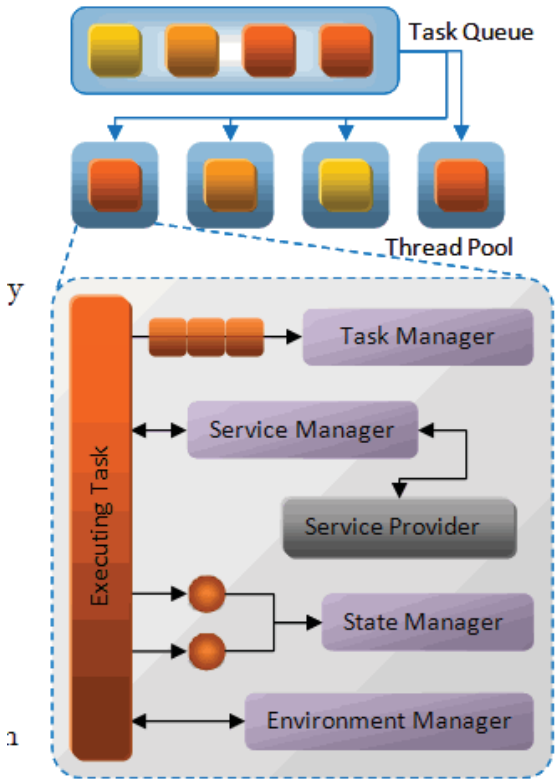


Figure 12. The task manager and tasks.

Distribution

Once all tasks targeted for the current clock have completed execution, the main loop calls the state manager to distribute the changes.

1. The state manager calls each of its change controllers to distribute the changes they have queued. This is done by going through each subject's changes and seeing which observer was listening to that subject.

2. The change controller then calls the observer informing it of the change (a pointer to the subject's interface is also passed to the observer). For free step mode the observer gets the changed data from the change controller, but for lock step mode the observer queries the subject for the data.

3. The observers that are interested in the changes done by a system object will typically be other system objects that are attached to the same universal object. This makes it possible for the change distribution to be broken up into tasks for execution in parallel. To limit synchronization, any universal objects' extensions that are linked should be grouped together in a task.

Runtime Check and Exit

The final step of the main loop is to check the runtime's state, which might be run, pause, or next scene. If the runtime state is set to run, it will repeat the entire game loop. If the runtime is set to exit, it exits the game loop, frees up resources, and exits the application.

Final Thoughts

Designing systems for functional decomposition, coupled with data decomposition, will deliver a good amount of parallelization and also ensure scalability with future processors that have an even larger number of cores. Remember to use the state manager along with the messaging mechanism to keep the data in sync with only minimal synchronization overhead.

The observer design pattern is a function of the messaging mechanism, and some time should be spent learning it. With experience, the most efficient design possible can be implemented to address the engine's needs. After all, the observer design pattern is the mechanism of communication between the different systems to synchronize all shared data.

Tasking plays an important role in proper load balancing. Follow the tips in Appendix D to create an efficient task manager for your engine

Designing a highly parallel engine is a manageable task if you use clearly defined messaging and structure. Properly building parallelism into your game engine will give it significant performance gains on modern and future processors.

About the Author

Jeff Andrews is an Intel application engineer currently focused on PC gaming. He is working on optimizing code for software developers and researching different technologies to enhance performance or add new features to games.

Over the years he has written several papers on optimization, threading, and gaming, including "Threading Basics for Games" and "Threading the OGRE 3D Render System." His most recent role was lead architect for Intel's Smoke game demo framework.

Jeff graduated with a B.S. in Computer Science from De La Salle University in Manila, Philippines (http://www.dlsu.edu.ph/).

When not working or playing games (preferably from the adventure genre), he is involved in activities with his church.

Appendix A.

Example of an Engine Diagram

This diagram gives an example of how the different systems are connected to the engine. All communication between the engine and the systems goes through a common interface. Systems are loaded via the platform manager (not shown).

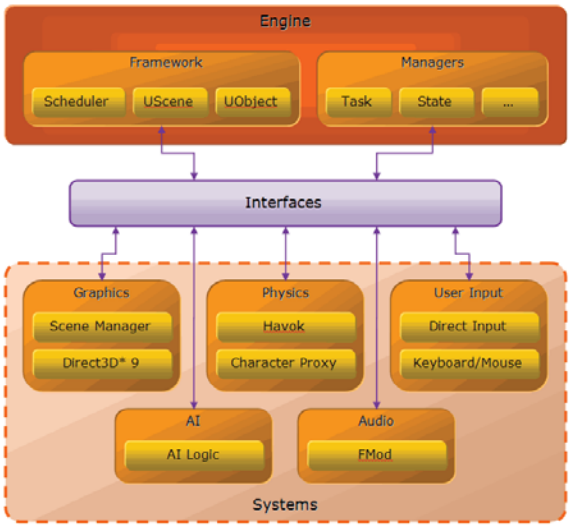
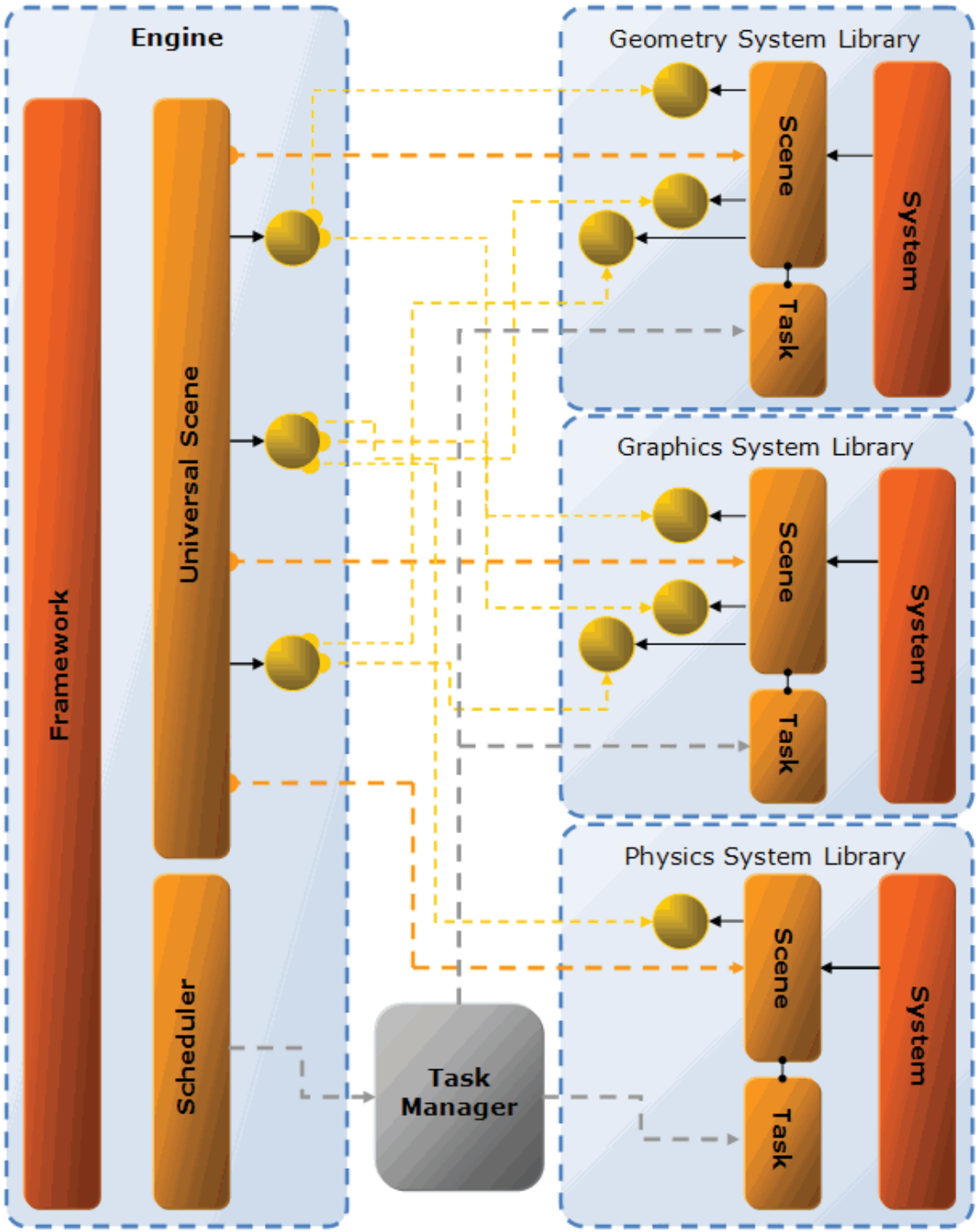


Figure 10. The engine manager and system initializations.

Appendix B.

Engine and System Relationship Diagram



Appendix C.

The Observer Design Pattern

The observer design pattern is documented in *Design Patterns: Elements of Reusable Object-Oriented Software*.

With this pattern, any items interested in data or state changes in other items do not have to poll the items from time to time to see if there are any changes. The pattern (Figure 13) defines a subject and an observer that are used for the change notification—the *observer* observes a *subject* for any changes. The *change controller* acts as a mediator between the two.

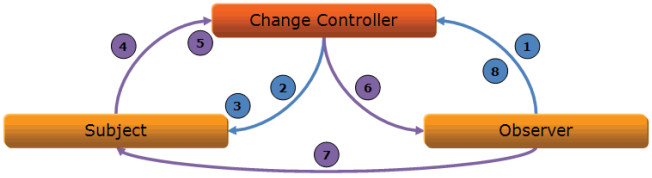


Figure 13. The observer design pattern.

1. The observer, via the change controller, registers itself with the subject for which it wants to observe changes.
2. The change controller is actually an observer. Instead of registering the observer with the subject it registers itself with the subject and keeps its own list of which observers are registered with which subject.
3. The subject inserts the observer (actually the change controller) in its list of observers that are interested in it; optionally there can also be a change type that identifies what type of changes the observer is interested in—this helps speed up the change notification distribution process.
4. When the subject makes a change to its data or state it notifies the observer via a callback mechanism and passes information of the types that were changed.
5. The change controller queues the change notifications and waits for the signal to distribute them.
6. During distribution the change controller calls the actual observers.
7. The observers query the subject for the changed data or state (or get the data from the message).
8. When the observer is no longer interested in the subject or is being destroyed, it deregisters itself from the subject via the change controller.

Appendix D.

Tips on Implementing Tasks

Although task distribution can be implemented in many ways, try to keep the number of worker threads equal to the number of available logical processors of the platform. Avoid setting the affinity of tasks to a specific thread, because the tasks from the different systems will not complete at the same time. Specific affinities can lead to a load imbalance among the worker threads, effectively reducing your parallelization. Also, consider using a tasking library, such as Intel® Threading Building Blocks, which can simplify task distribution.

Two optimizations can be done in the task manager to ensure CPU-friendly execution of the different task submitted.

- **Reverse Issuing**. If the order of primary tasks being issued is fairly static, the tasks can be alternately issued in reverse order from frame to frame. The last task to execute in a previous frame will more than likely still have its data in the cache, so issuing the tasks in reverse order for the next frame will all but guarantee that the CPU caches will not have to be repopulated with the correct data.
- **Cache Sharing**. Some multi-core processors have their shared cache split into sections, so that two processors may share a cache, while another two share a separate cache. Issuing sub-tasks from the same system onto processors sharing a cache increases the likelihood that the data will already be in the shared cache.

Bibliography

Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software. USA*: Addison-Wesley, 1994.

Intel® Threading Building Blocks (TBB). Available from: http://www.threadingbuildingblocks.org/

Intel and Gamasutra – Visual Computing. Available from: http://www.gamasutra.com/visualcomputing/

Multi-threaded Game Programming and Hyper-Threading Technology. Available from: http://software.intel.com/en-us/articles/multithreaded-game-programming-and-hyper-threading-technology

Reinders, James. *Intel Threading Building Blocks*. USA: O'Reilly Media, Inc., 2007.

Smoke – Game Technology Demo. Available from: http://software.intel.com/en-us/articles/smoke-game-technology-demo

Threading Basics for Games. Available from: http://software.intel.com/en-us/articles/threading-basics-for-games

Threading Methodology: Principles and Practice. Available from: http://software.intel.com/en-us/articles/threading-methodology-principles-and-practice

Printed in USA 0109/CS/RHM/PDF