A CONCURRENCY MODEL FOR GAME SCRIPTING

Joseph Kehoe
Institute of Technology, Carlow
email: joseph.kehoe@ITCarlow.ie

Joseph Morris
Dublin City University
email: joseph.morris@computing.dcu.ie

KEYWORDS
Concurrency, Games, Scripting

ABSTRACT

In this paper we outline a new model of concurrency that is specifically designed for the specialized domain of game scripting. Scripting is used extensively in game development both for the implementation of AI based behaviors and for providing game players with the ability to customize commercial games. Scripting languages have not, as yet, benefited from the move to multicore architectures. We discuss the properties unique to game scripting that any proposed model must satisfy. Then we propose a model of concurrency particular to games that addresses these issues while allowing game scripting languages to fully utilize multicore processors. The next steps in developing this model further are discussed.

INTRODUCTION

Game scripting is an integral part of computer game development. Scripting is essential for two reasons, the nature of the game development process and the type of game developers who write game behaviors.

The game development process is an iterative one. This is particularly true for entity behavior design and implementation. Entities in games are any objects that can interact with their surroundings and the player of the game. They range from pretty trivial items such as doors to non player characters that can form their own plans. Game entity behaviors can really only be properly assessed by actually playing the game with the entity behaviors *in situ*. This leads to the employment of rapid *design-implement-playtest* iterations. For this to be a viable process each iteration needs to be as short as possible. It can take many iterations before a behavior becomes acceptable within a particular game. These rapid iterations presume the use of a scripting language. Scripting languages are high level languages that allow for rapid implementation (or prototyping) of behaviors. As these languages are almost always interpreted rather than compiled they can be rapidly deployed, usually without requiring a recompilation of the surrounding game framework.

The second aspect of the development process that is applicable here is the programming ability of game designers. Behavior design is the domain of game designers in that they know best what behaviors are suitable for each game scenario. Game designers, however, are not game programmers. They do not have the full range of programming skills that are available to professional programmers.

Forcing game designers to pass their behavior designs onto professional programmers for implementation is not a viable option. It would tie up an expensive professional programmer who could be employed on other parts of the game while also slowing down the *design-implement-playtest* cycle by an unacceptable amount of time. Ideally, the game designer herself, as the person who fully understands the required behavior, should be able to implement the behaviors directly. To make this possible a simple high level scripting language is required.

Processor power (and speed) has been increasing at an almost constant rate, following Moore's law, since the introduction of the integrated circuit. Future increases in speed will be through the use of multicore processors, (Blake et al. (2009)). More speed will mean more cores on each processor. Processors which used to contain a single complex core will now be replaced by processors containing many cores (J. Held and Koehl (2006), Borkar et al. (2006)).

The only way for software to take advantage of these new architectures is by simultaneously using as many of the cores as possible. In other words software must switch from being sequential to concurrent. Unfortunately, writing concurrent code is more difficult than writing sequential code, particularly when using programming languages designed mainly for sequential programming. New techniques for designing, writing and testing concurrent software are needed and new programming languages may also be required.

Our model of concurrency is specific to scripting in the game development process. Scripting in games poses some unique challenges. We have two competing issues. Firstly, games are real time systems with hard time constraints. This implies the use of a low level language where developers have complete control over hardware resources. Secondly, and in opposition to this, it involves the use of a high level scripting language that will hide the details of the hardware from the programmer. Real time programming is a specialist skill that is acquired by highly competent programmers only after

many years experience while scripting in games must be open to non professional programmers.
Game developers are willing to accept the usage of scripting languages in the development of games for the reasons given in the previous subsections. Using concurrency to help improve the efficiency of scripting languages is one way of tackling this issue.

Overview of paper

In the next section we give an overview of the model we propose. We look at key features of this model of concurrency and show how it fulfills the expectations set out in the previous section. We follow by reviewing related work in concurrency and game scripting. Finally we finish with our conclusions and list further work that needs to be completed.

PROPOSED CONCURRENCY MODEL

A game is a simulation that consists of a set of entities interacting with each other in some world. Each entity has its own state and a set of behaviors that determines how it responds to various stimuli. The game world that is being simulated has rules (gravity for example) that determine how certain types of interaction between entities take place. These world rules can be encoded in the entities themselves. The global state of the game is given by the sum of the states of all the entities that it contains.
Games proceed in a stepwise manner as a sequence of discrete moments in time. Each step represents a tick of the clock, or one particular moment in time, and a sequence of steps represents the passing of some duration of time. The step frequency can vary, with the frequency representing the granularity of the representation of time in the game. Step frequency is ultimately determined by hardware factors such as processor speed. At each step the entities update their state based on the events or stimuli that were generated during the previous step. Overall entity behavior through the lifetime of a game comprises the sequence of step behaviors made during that game. A play-through of a game is a finite sequence of these steps.
In the next sections we will look at the basic structural components of the model and how they fit together. These components are: entities, messages, steps and constraints.

Entities

The entity is the basic building block of game simulations. Games consist of many different entities that interact with each other under some game defined rules. These rules may include, for example, physics based interaction such as gravity and elasticity. Entities range from the simple, like a bullet or item of furniture, to the more complex such as a non player character (NPC) that has its own beliefs, desires, intentions and plans of action.
In general, although entity types are varied we can assume that they have some common attributes such as geographical position in the game world, an associated model (a visual 3-D representation of the item) and boundary dimensions (used for collision detection). More complex games will also model mass, elasticity and internal structure (a skeleton) for each entity as well. Everything can be defined in terms of individual entities and their rules of interaction. In our model only entities exist in games.
Game design consists of (among other things not relevant here such as ensuring game playability) identifying all the entities that make up a game, deciding which of their properties are relevant to the game and how they are allowed to interact.
Every entity runs in its own thread. Concurrency is limited only by the number of entities in the game. Games are simulations designed to be fun to play. Given the nature of games it is unlikely that there will be only a few entities in a game. Since the identification of entities within a game is already part of game design this approach does not add any extra overhead to game design. It is a natural way to identify concurrency and brings identification of concurrency easily within the remit of the game designer at no extra cost.
Every entity is composed of four components: state, interface, constraints and message queue.
State is a non-empty set of named attributes. Each attribute will have an associated value. A non-empty subset of these attributes will be immutable. Immutable attributes are given values when the entity is created and these values remain unchanged until the entity is destroyed. Each attribute is also labelled as either visible or hidden. The full set of named attributes with associated labels belonging to an entity is called the entity state. The particular value of a state is determined by the values assigned to each attribute in the state.
Each entity contains at least one attribute as part of its state. This is the ID attribute which uniquely identifies the entity. This is both immutable and visible. The value of the ID attribute is generated automatically on entity creation and guaranteed to be unique for each entity.
Entities know a non empty subset of the state belonging to every other entity. This subset will contain all the attributes that are labelled visible. If an entity knows the attribute of another entity then it is allowed to read the value contained by that attribute but it cannot write to a attribute belonging to the state of any other entity. Changes to an entity attribute value can only occur through the entity interface.
Every entity has a defined set of message signatures. Each message in this set represents the response of the entity to a specific type of stimulus. Each stimulus event

triggers a specific message for each entity affected by that stimulus. An entity can only respond to a stimulus if that stimulus generates a message matching the signature of a message defined in its interface. The signature of a message is determined by a message name and a sequence of attributes.
Each signature must be unique within an interface. The sequence of attributes is non empty and will include at least the sender attribute. For any message instance the value of this attribute will match the value of the ID attribute of the entity that generated the message. The set of message signatures is known as the interface. Entities know the full interface of every other entity. An entity is allowed send a message to any entity if it knows the value of that entities ID attribute.
The set of local constraints determines the set of acceptable combinations of values that the attributes belonging to the entity state can hold.
The constraint set may also determine allowable combinations of values that different entities can simultaneously hold. That is, the allowable values of an entity state can be determined by the values of other entity states. For example, it may be the case that two entities cannot occupy the same position in space simultaneously.
The message queue contains the full set of outstanding messages that the entity has to respond to during the current step. This queue will contain all messages generated for the entity during the previous step. Messages are processed by each entity in the order in which they appear in the message queue. Each entity message queue can only hold messages that match message signatures defined in that entities interface. When an entity is destroyed its associated message queue and any remaining messages in that queue are also destroyed.

Messages

A message consists of a name, a receiver ID and a collection of attributes and the values of those attributes. A message generated (or sent) during one step will always be received during the next step. This guarantee that all messages are processed during the succeeding step means that, as a consequence, we cannot fix an upper limit on step duration in advance without limiting the number of messages allowed to be generated in each step. As the number of messages increases step duration will also increase. A fixed step duration would be an advantage for any real time system but it has a number of associated costs.
Most importantly by fixing step duration we would lose computational determinism. Different processors are able to accomplish different amounts of work in the same duration. The overall result of any step would then become dependent on the processor speed. We maintain that determinism is more important than fixed step duration.
Determinism gives us independence from the underlying processor executing the scripts. This greatly simplifies the testing and debugging of scripts. If testing and debugging depended on the processor used it would become beyond the capabilities of many scripters and reduce the possibility of using the model in the high level prototyping and rapid iterative development cycles used in game development. Since we can decouple the step rate from the frame rate any increase in step duration can be handled in a graceful manner by the game engine.
The value of the Receiver ID is used to determine who the receiver of the message should be. Each entity has an associated message queue and messages are put in the queue belonging to the entity whose ID attribute value matches that of the receiver ID in the message. Messages are processed in the order that they appear in the message queue.
A message is acceptable only if it fulfills two conditions Firstly, the receiver ID value matches the ID attribute value of an existing entity and secondly that the message name and attribute collection matches a message signature defined in the interface of the entity whose ID matches that of the receiver ID.
All unacceptable messages are discarded. Only acceptable messages appear in message queues. In response to a message an entity can do any or all of the following:

1. Send messages to other entities if it knows the values of those entities ID attribute;
2. Create one or more new entities;
3. Update any of its own mutable attributes provided that these updates do not violate any of the constraints in its constraint set;
4. Destroy itself.

Steps

A complete computation consists of a sequence of two or more steps where the first step is the initialization step, the last step the shutdown step and all other steps are intermediate steps. During the initialization step two processes occur: entities are created and initial Messages are generated. The final step consists of two parts: the remaining messages are discarded and finally all entities are destroyed.
For every intermediate step all acceptable messages that were generated during the previous step are processed. All acceptable messages generated during the previous step will be present in the appropriate message queues at the start of the current step.
Entity state is updated instantaneously and simultaneously at the end of each step. State update is defined as the sequential composition of the messages contained in the message queue, in order, modified by a conflict resolution algorithm.

Any messages generated during this process are delivered instantaneously at step end. Delivered messages are put in the message queue belonging to the entity whose ID value matches that of the receiver ID value in the message.
The order that messages are placed in the message queue is defined by the message sorting algorithm. The message sorting algorithm can be any algorithm that guarantees:

1. Messages generated by a single entity for the same receiver are placed in the message queue in the same order that they were generated in;

2. The final order of the message queue is deterministic. That is, for any given set of messages their ordering is unique and will always be the same regardless of how many times the ordering algorithm is applied.

The default sorting algorithm orders message queues by using the message Sender ID attribute as the primary key and message generation order as the secondary key. This ensures that both conditions hold. Any other algorithm that fulfills our two guarantees is acceptable. The message sorting has an associated cost. Between steps sorting will have to be carried out. This overhead is justifiable because the algorithm ensures determinism in script execution.
Determinism is important because it isolates the script and the scripter from the underlying processor architecture. As we have already stated this makes testing and debugging feasible in the rapid development cycles used in game development and also in the prototyping environment of casual and hobbyist game development.

Conflict Resolution

That conflicts can arise between different entities is a recognized problem in games. Entities exist in a common world. In this world rules will exist that govern how these entities can interact with each other. Attempts to perform certain actions will bring entities into conflict with these rules. There are three possible approaches to conflict resolution:

1. Put onus on scripters;

2. Handle constraints using other parts of the game engine;

3. Let scripting system handle constraints automatically.

The first option is, in many ways, the simplest. The scripter should ensure that any code they write does the proper error checking. This has the advantage that it can be the most efficient technique. Scripters will know when run time checks are required and when they are not. Although this is a common approach it has the disadvantage of putting the burden on the individual scripter. Under our model there is the extra complication that entities can only see the state of other entities as they were at the start of the step making it difficult to check for conflicts with other entities during a step.

The second option is suitable when the constraints logically fall within the remit of some other specialized subsystem. This is the case when the constraints between entities are real world constraints. In this case the game physics engine can handle the constraints very efficiently.

The final option covers cases where the first two options are judged unsuitable. We use a predefined conflict resolution algorithm to determine how conflicts are dealt with. The conflict resolution algorithm is any well defined algorithm that ensures that state update does not violate any constraints defined in the entities constraint set. Constraints can be divided into two different types: *internal* and *external* constraints. Internal constraints are constraints that exist only within a particular entity. External constraints are constraints that hold between two or more different entities.

Internal, or local, constraints are the easier to deal with than external constraints. They are, by definition, internal stand-alone constraints and so each entity can deal with them independently of any other entity. Internal constraints are state invariants that must hold throughout the entity lifecycle. These constraints are defined at entity creation and can be checked after each message is processed. If a conflict is detected the algorithm can take the appropriate corrective action.

The default algorithm simply discards any messages that cause conflicts during message composition. Other algorithms may be employed that take different error correction measures.

External conflicts can only be detected once the step is completed. This is because we cannot tell the final state of each entity until the end of the step. Once all entities have completed their state update the value of each entity state needs to be checked for conflict with every other entity. Once a conflict is found between two or more entities a state rollback, of some predetermined kind, of one or more of these entities will be required. After rollback of an entity state we may have to recheck the new value of the state against the global constraints. This has the potential to be more costly than internal constraint checking as one state rollback can raise more new conflicts.

We do not feel that external constraints will be common or form an essential part of any game script. Firstly, most external constraints will be handled independently by the physics engine. Secondly, remaining external constraints can be encoded as one or more equivalent internal constraints. For these reasons we do not have a separate default algorithm to handle external constraints.

RELATED WORK

BSP - Bulk Synchronous Processing

BSP has been proposed as a bridging model for general purpose parallel computation by Valiant (1990). A BSP computation consists of a sequence of super-steps. In a super-step each component (a processor or core) is allocated a task consisting of a combination of local computation and, message transmission and reception from other components. After L time units have passed a check is made to see if the super-step has completed. If it has, then the next super-step is started. Otherwise the next L units are allocated to completing the current super-step.
In simple terms, at each step a set of local computations is undertaken concurrently. According to Skillicorn and Talia (1998), the aims of BSP are to make it simple to write concurrent code, be independent of target architectures and make performance of a program on a given architecture predictable. BSP allows you to put an upper time bound on a computation for a particular architecture. This makes the performance more predictable. In addition, deadlock using BSP is impossible. BSP is also easier to debug in that computations can be rearranged inside a superstep without affecting the outcome. BSP has been successfully integrated in scripting languages in the past by, for example, Hinsen (2007).

COOP - Concurrent Object Oriented Processing

Three types of concurrent object model have been identified: Orthogonal, Homogeneous and Heterogeneous (Papathomas (1995)).
The Orthogonal model views the object model and the concurrency model as two separate independent systems. In this case, locks are used to resolve any issues raised by concurrency. The orthogonal approach does not gain us any ground as it still retains explicit locking and all the problems that this implies (Sutter and Larus (2005)).
In the Homogeneous approach all objects are *active objects*. An active object is an object that runs inside its own thread. It represents a merging of process and object (Briot et al. (1998), Hernandez et al. (1994)). The internal state of an active object is private to that object. Any interaction that must take place between objects must take place via message passing. Generally, messages are asynchronous but there is variation between explicit or implicit acceptance of messages by objects.
The heterogeneous model contains both the active objects of the homogeneous approach and the passive objects of the orthogonal model. The most popular form of concurrent object oriented programming model is based on active objects.
Actor models of concurrency are closely based on active objects. An Actor is an active object that can send finite set of messages to other actors, create a finite set of new actors and define how it will behave in relation to the next incoming message. - Corrêa (2009)

Network Scripting Language

The Network Scripting Language described in Russell et al. (2008) (NSL) is designed for distributed games development. It runs across remote processors rather than multiple cores and gives some assurances of determinism and consistency maintenance between the various processors during game execution.
NSL uses active objects and a frame based approach similar to the approach advocated here. Because this language is designed for multiple distributed processors each processor will have its own copy of the state of the objects in the other processors. If there are n processors then there will be n copies of the overall state.
This approach, out of the three mentioned, is the most similar to our approach but the programming language is more complex. It is tightly coupled to the frame rate with a step being run exactly once for every frame but gives no guarantees as to when messages will be delivered.

CONCLUSION

We have outlined a model of concurrency developed specifically for games development, specifically game scripting. Game scripting is undertaken by game designers who are not professional programmers and therefore do not have an in-depth understanding of concurrency. As the game entity behaviors they develop have to be play tested to ensure they are appropriate they must use many rapid *design-implement-playtest* iterations during development. To enable them to make use of concurrency we developed a model that is easy to use, removes as much of the burden from the designer as possible and can be implemented in any standard game scripting language.

Further Work

Some work remains to be done on developing algorithms that can be used by the conflict resolver. Although a simple conflict resolution algorithm has been proposed it may be the case that different games will need to use different or more sophisticated conflict resolving algorithms.
We intend to produce a working implementation of our model. This implementation will demonstrate the viability of this approach. It will be incorporated into an existing game scripting language to show how it fits into already existing development tools and practices in the games industry. This will also serve to show how transparent this model is to game scripters in practice. As

well as demonstrating how simple the model is to use it will also show how easy it is to incorporate into existing game scripting languages.

Biography

Joseph Kehoe is a lecturer in Computing in the Institute of Technology Carlow. He has previously been director of the BSc in Games Development and is currently Director of the BSc in Software Development.
Professor Joseph Morris is a lecturer in Computing in Dublin City University.

REFERENCES

G. Blake, R. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26 – 37, november 2009. ISSN 1053-5888. doi: 10.1109/MSP.2009.934110.

S. Borkar, H. Mulder, P. Dubey, S. Pawlowski, K. Kahn, J. Rattner, and D. Kuck. Platform 2015: Intel processor and platform evolution for the next decade. Intel White Paper, 2006.

J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998. ISSN 0360-0300. doi: http://doi.acm.org.remote.library.dcu.ie/10.1145/292469.292470.

F. Corrêa. Actors in a new "highly parallel" world. In *WUP '09: Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*, pages 21–24, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-565-9. doi: http://doi.acm.org.remote.library.dcu.ie/10.1145/1527033.1527041.

J. Hernandez, P. de Miguel, M. Barrena, J. Martinez, A. Polo, and M. Nieto. Parallel and distributed programming with an actor-based language. In *Parallel and Distributed Processing, 1994. Proceedings. Second Euromicro Workshop on*, pages 420 –427, 26-28 1994.

K. Hinsen. Parallel scripting with python. *Computing in Science and Engineering*, 9(6):82–89, 2007.

J. B. J. Held and S. Koehl. From a few cores to many: A tera scale computing research review. Intel White Paper, 2006.

M. Papathomas. Concurrency in object-oriented programming languages. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 31–68. Prentice Hall, 1995.

G. Russell, A. F. Donaldson, and P. Sheppard. Tackling online game development problems with a novel network scripting language. In *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 85–90, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-132-3. doi: http://doi.acm.org.remote.library.dcu.ie/10.1145/1517494.1517512.

D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/280277.280278.

H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005. ISSN 1542-7730. doi: http://doi.acm.org.remote.library.dcu.ie/10.1145/1095408.1095421.

L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990. ISSN 0001-0782. doi: http://doi.acm.org.remote.library.dcu.ie/10.1145/79173.79181.