Birds on the Wall: Distributing a Process-Oriented Simulation

Adam T. Sampson, John Markus Bjørndalen and Paul S. Andrews

***Abstract*— The CoSMoS project aims to develop reusable tools and techniques for complex systems modelling and simulation. Using process-oriented software design techniques, we have built a concurrent model of continuous space, usable in a variety of complex systems simulations. In this paper, we describe how we refactored our space model to allow our simulations to run in an efficient and highly-scalable manner across clusters of commodity machines—and, in particular, to support distributed simulation and visualisation on the Tromsø Display Wall.**

I. Introduction

The CoSMoS project[1] aims to develop a common framework for the modelling and simulation of complex systems using process-oriented software design, including a modelling process, a set of design patterns and refactoring procedures, and a toolkit of reusable software components.

CoSMoS takes a case-study-based approach to the development of modelling techniques. We have implemented a variety of textbook and real-world complex systems using process-oriented approaches, including ant colonies, bird flocking, chemical diffusion, small-world networks, symbol rewriting systems and a variety of immunological models. In addition, the TUNA project (a pilot for CoSMoS) implemented simulations of cellular automata [1] and haemostasis [2]. From these case studies, we extract design patterns that can be applied to solve common problems in complex systems modelling and simulation.

One well-known complex system is Reynolds' boids [3], a simulation of emergent flocking behaviour in birds. Boids are independent agents moving around in continuous N-dimensional space; we use two-dimensional space here for simplicity. They have a limited field of view (an arc with a fixed radius immediately in front of them; see Figure 1), and follow a set of simple rules to decide their behaviour based upon the other agents they see:

- move toward the centroid of the visible flock;
- match the mean velocity of the visible flock;
- move away from other agents if they are *very* close.

These rules cause flocks to form as an emergent behaviour. Boids is typically implemented by having each rule compute a force acting upon the boid; the forces are summed to produce the boid's velocity at each timestep.

Boids is of particular interest to CoSMoS as a case study because it includes a number of common elements in complex system models: boids must move in continuous space, they must be aware of other agents in their neighbourhood; and they may react differently to different types of agents. In addition, the boids model can be easily modified by the addition of new rules (for example, to follow a particular path, or to react to predators or food); it operates well at a variety of levels of scale (from a handful of boids up to tens of thousands), and it has a straightforward visualisation in which anomalous behaviour can be easily spotted.

A *display wall* is a large, high-resolution display system for scientific applications, built by tiling a number of smaller displays [4]. The Tromsø Display Wall [5] consists of 28 projectors arranged in a 7x4 matrix, back-projecting onto the wall of a meeting room. The resulting display has a total resolution of 7168x3072 pixels (22 megapixels), and a diagonal size of 230". Each projector is driven by a dedicated PC, with the resulting 28-node cluster is connected by gigabit Ethernet. As the Display Wall was constructed entirely from commodity components, its overall cost was relatively low.

The Display Wall [5] is an ideal system for visualising large-scale simulations such as those built by CoSMoS. While many existing Display Wall applications Wall have a simulation running elsewhere and use the cluster hosts only for distributed visualisation (or even as relatively-dumb displays), we would like to take advantage of the computational power available in the cluster to perform a distributed simulation with local visualisation. In this paper, we will describe the process-oriented model of space that we have developed for simulations of systems such as boids, and how we have modified it to run efficiently on clusters of machines such as the Display Wall.

Section II gives a brief introduction to process-oriented programming, and the facilities we have used in our simulation. In section III, we describe our existing model of space upon a single host. Section IV describes the refactoring process that we applied to distribute our simulation across

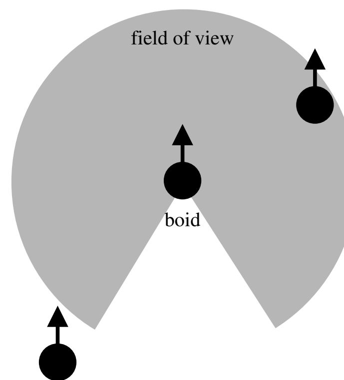

Fig. 1. Boids in two-dimensional space.

Adam Sampson is with the Computing Laboratory, University of Kent, UK (email: ats@offog.org). John Markus Bjørndalen is with the Department of Computer Science, University of Tromsø, Norway. Paul Andrews is with the Department of Computer Science, University of York, UK.

[1] http://www.cosmos-research.org/

978-1-4244-2959-2/09/$25.00 ©2009 IEEE

multiple hosts. In section V, we discuss the problems arising from distributed visualisation upon the Display Wall. Finally, section VI gives our conclusions and plans for future work.

II. Process-Oriented Programming

The CoSMoS approach to software engineering is based upon *process-oriented programming*, which emphasises the construction of isolated, concurrent, lightweight *processes* that only interact using clearly-defined interfaces. The core ideas of process-oriented programming derive from the CSP [6] and π-calculus [7] process algebras, making it possible to reason formally about the behaviour of individual processes and the system as a whole.

A number of software libraries and specialised programming languages are available to support process-oriented programming in a safe, efficient way; in particular, the CoSMoS project makes use of occam-π [8] and JCSP [9]. occam-π is a concurrent procedural language with support for extremely lightweight communication and synchronisation between processes; the occam-π compiler performs a variety of static checks to detect common concurrency errors at compile time. JCSP is a library for Java that provides process-oriented programming facilities on top of Java's native threading model; it allows process-oriented systems to be constructed that can make use of and integrate with existing Java code.

We believe that process-oriented programming is an effective approach for complex systems simulation because many complex systems are themselves inherently highly concurrent. Using a process-oriented approach, concurrent entities such as agents can be modelled directly as processes. The resulting simulation expresses a high degree of parallelism, with agents able to execute their behaviours concurrently. With appropriate runtime load-balancing support—such as that provided in the KRoC occam-π implementation [10]—a process-oriented system can take direct advantage of multi-core CPUs and other parallel hardware.

Many process-oriented programs make use of the *client-server pattern* [11], in which server processes respond to requests from client processes. Servers and clients are joined using *client-server connections*, typically implemented using pairs of communication channels. While a server may only deal with one client at a time, server interfaces can be shared, allowing multiple clients to compete for access to the same server. Clients and servers may have arbitrarily long conversations over a connection with several messages sent in either direction. A server may itself act as a client to other servers while fulfilling the client's request; the structure of a client-server system is typically a tree.

The set of messages that may be exchanged on a particular client-server connection is described by a protocol [12]. Compile-time checks ensure that processes follow their protocols. The client-server design rules describe how to construct client-server relationships between processes so as to ensure that the resulting system will be free from deadlock and livelock. Server processes are often used in process-oriented programming as a safer alternative to objects located in shared memory. We design client-server systems (and process networks in general) using an informal graphical language; see Figure 2 for an example.

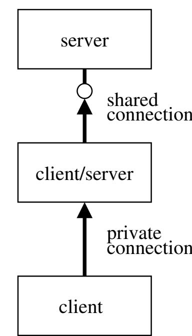

Fig. 2. Client-server relationships between processes.

Barriers are often used to provide all the agents in a simulation with a shared sense of time. A barrier is a synchronisation object visible to a number of processes. When a process tries to synchronise upon the barrier, it will be blocked until all the processes that can see the barrier are also trying to synchronise upon it; once all processes are engaged, the barrier completes and the processes can continue. (A barrier is therefore equivalent to a CSP event.)

The simplest way to regulate simulation time using barriers is to have each process synchronise on a shared barrier at the end of each timestep; that way, no process can enter the next timestep until all the others have finished the previous one. While this is sufficient for the boids simulation, it is somewhat inefficient in that every process must take part in every timestep. More elaborate schemes can be used to provide for multiple timescales or more flexible timing requirements within a more complex simulation.

It is often useful to subdivide each timestep into several *phases* [13], with a barrier synchronisation by all interested processes at the end of each phase. A model with agents viewing an environment could use two phases: in the first phase, all the agents would obtain a view of their surroundings, and in the second phase, they would perform their movement actions for the current timestep. This approach ensures the consistency of the simulation—the view will not change before all the agents have obtained it—while still allowing the agents' computations to proceed in parallel.

III. Modelling Space

Occoids, our implementation of the boids model, is written in occam-π. It features two types of agents: boids, which follow the boids rules, and trees, which have no behaviour and simply act as obstacles for the boid flocks to navigate around.

A. Modelling Position

While discrete positioning is sufficient for systems such as cellular automata, in order to simulate a system like boids it is preferable to allow coordinates to be real numbers. We

model space as a regular grid of *location* server processes, each of which represents a region of continuous space which may contain any number of agents [14]. Each agent is a client to the location it occupies. Dividing our space model into multiple processes allows agents to find out about nearby agents while avoiding contention on a centralised store of agent positions.

A location keeps a record of the agents it currently contains, along with their positions relative to the centre of its region. Each location is provided with references to the server interfaces of its eight neighbouring locations (although it does not communicate with them directly itself). Agent positioning is therefore entirely relative, with no inherent need for a global coordinate system; this allows space to be dynamically extended, and unusual spatial topologies to be constructed.

The interface provided by location processes supports three requests:

- *enter*, sent by an agent when it first enters a location along with an initial position;
- *move*, sent by an agent when it wants to move, along with a velocity vector that will be added to its current position;
- *look*, which responds with a list of the agents in the location.

The server will respond to *enter* and *move* requests with one of two responses:

- *stay-here*, if the agent can stay in the current location;
- *go-there*, if the agent has moved across a location boundary, along with a reference to the new location's server interface and a position relative to that location's centre.

In order to move, an agent sends a *move* message to its location giving its velocity. If the server replies *stay-here*, no more needs to be done. If it replies *go-there*, the agent must send an *enter* message to its new location. The new location may in turn reply either *stay-here* or *go-there*; the agent simply repeats this process until it receives a *stay-here* response. This allows an agent to move across several locations in one step, without the locations needing to know about anything other than their immediate neighbours.

B. Modelling Vision

At the start of each timestep, each boid must look around to find the other agents within its field of view. One way to achieve this would be for each boid to directly interrogate all the locations that intersect its field of view—but this is inefficient, because each boid must perform several communications, and awkward because it requires each boid to be a client to all the locations it can see into as well as the location it is in.

Instead, each location has a corresponding *viewer* process, which at the start of each timestep interrogates the surrounding location processes and builds a *view list* containing all the agents that may be visible to an agent inside its location, and their positions relative to the centre of its location. (At present, we restrict the maximum radius of a boid's field of view to be the width of a location, so the viewer process only needs to interrogate its location and its immediate neighbours.) We split each timestep into two phases to ensure that the viewer processes have all obtained a consistent view of the world before the agents start moving around.

The viewer provides a server interface that responds to a *look* message with its current view list—which the agent must filter to extract only the other agents that are actually within its field of view, and to exclude itself. We extended the location protocol so that when an agent enters a location, it is given a connection to the corresponding viewer. The viewer is therefore shared between all the agents in a location, and the view needs only to be assembled once per timestep.

C. Modelling Agents

Many simulations need to include multiple types of agents with different behaviours. To make this easier, we divide the responsibilities of the agent as described above into a pair of processes: an *agent* process and a *behaviour* process, joined by a private client-server connection.

The agent process implements the movement and viewing algorithms as described above, and provides a simple server interface that understands the following requests:

- *move*, with a velocity vector, producing no response;
- *look*, which causes the agent to respond with the current view.

The behaviour process implements the behaviour appropriate for the agent's type. A boid's behaviour process repeatedly retrieves the view with a *look* request, applies the boids rules to the result, and sends a *move* request with the computed velocity. A tree's behaviour process does nothing.

Using this approach, the details of the space model are entirely hidden from the behaviour process. This makes it straightforward to add new types of agents to the simulation, since the programmer only needs to implement a new behaviour process and can reuse the existing agent process. Furthermore, we can change the implementation of the space model by changing only the agent process.

Process creation is cheap in process-oriented systems, so the overheads of abstracting common behaviour out into a separate process in this way are minimal—comparable to abstracting shared behaviour out into a superclass in an object-oriented system.

D. A Complete Simulation?

Figure 3 shows a sample process network for the single-host simulation. An overview of the protocols and phases used in the simulation are shown in Figures 4 and 5 respectively.

The simulation we have described is highly concurrent—agents compute their movement vectors in parallel—and owing to the occam-π runtime's load-balancing support, it can already take advantage of multiple CPU cores on a single host. However, in order to run bigger simulations, or to make use of distributed visualisation, we need to able to distribute the simulation across a cluster of hosts.

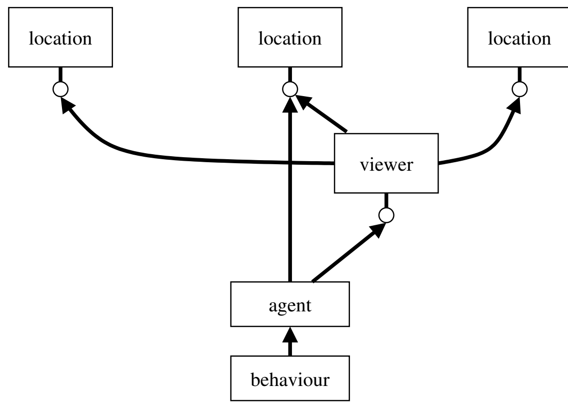

Fig. 3. An agent in one-dimensional space, in the single-host simulation.

IV. Distributing the Simulation

A. An Initial Approach

Our first approach was to distribute our existing the space model by splitting up our existing process network across multiple hosts in the cluster: each host therefore simulates a region of space by running the location processes corresponding to that region.

This can be achieved with relatively little effort using pony [15], a transparent networking system for occam-π. pony provides network channels that have the same semantics as local channels, but operate between hosts over a network. Network channels therefore have much higher latency than local channels. In addition, where local channels can simply move references to memory between processes upon communication, network channels must copy data. A local channel is automatically upgraded to a network channel if one end of it is sent over the network. In addition, pony provides a nameserver that lets a distributed occam-π application discover and assemble its components.

Network channels can be used to construct client-server connections over a network in the same way as local channels. Converting a local application to a distributed application using pony is usually very straightforward, since only the initial setup of the application's channels needs to be changed.

We modified our simulation so that it allocated network channels rather than local channels for the client-server connections between locations that are on opposite sides of a host boundary. The resulting simulation ran correctly—but very slowly. In addition, the performance became steadily worse as time progressed. This was the result of two problems:

- Firstly, on each timestep, all the viewer processes request the contents of the locations they are viewing. For viewer processes on the "edge" of a host, this means transferring the view over the network from one or more locations, rather than simply moving a reference locally. Furthermore, since each location is seen by multiple viewers, the view is transferred multiple times.
- Secondly, while boids move around in virtual space, their agent and behaviour processes remain on the host they started on—and are thus communicating with their locations and viewers over a high-latency, data-copying network connection. This is by far the more significant effect, and is the cause of the simulation slowing down as more boids migrate between hosts.

```
location
  = (enter | move(VECTOR))
      -> (stay-here | go-there(LOCATION))
  | look -> VIEW-LIST
viewer
  = look -> VIEW-LIST
agent
  = move(VECTOR)
  | look -> VIEW-LIST
```

Fig. 4. Protocols in the single-host simulation.

1. Viewer processes update
2. Agent processes retrieve views and perform actions

Fig. 5. Phases in each timestep of the single-host simulation.

B. Refactoring the Model

In order to get good performance from our simulation, we needed to refactor our model of space so that it presented the same behaviour to agents, while significantly reducing the number of network communications.

To fix the first problem, we introduced *ghost* processes. A ghost process acts as a local proxy for a location on the other side of a network connection, providing the same server interface as the location. (Ghost processes are therefore an instance of the *remote proxy pattern* [16] that is common in distributed applications.) In a new phase added at the start of each timestep, each ghost requests a view from its location and caches the result; when it receives a *view* request, it can respond immediately with the cached view without needing to consult the real location. Ghosts handle other requests by forwarding them to the underlying location.

In order to solve the second problem, we needed to move processes around between hosts: when a process attempts to cross a host boundary by moving to a location on a different host, it should be checkpointed, terminated on the current host, and restarted on the destination host. The occam-π language provides limited facilities for first-class suspendable processes in the form of *mobile processes* [17], but pony does not yet support sending mobile processes between hosts; fortunately, the same idea is relatively straightforward to implement by hand.

We extended the location protocol so that the *enter* message may elicit a *suspend* response. When the agent receives *suspend*, it generates a representation of its internal state, sends it back to the location in a *suspended* message, and then terminates. The location is then responsible for

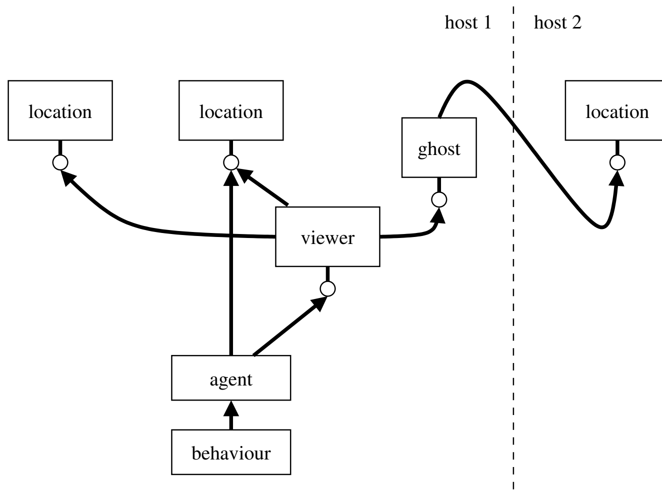

Fig. 6. One-dimensional distributed space, showing a ghost process.

restarting the agent using the saved state on the destination host—which will be the host that the location is running on.

Similarly, we extended the agent protocol so that suspension could happen upon a *move*. Unfortunately, the behaviour processes must be modified to support suspension; this is straightforward enough for simple agents such as boids that loop performing the same action, but would be more awkward for agents with complex internal control flow; better language support for explicit process suspension would make this more straightforward.

To detect when an agent is crossing a host boundary, we used the ghost processes. If an agent is trying to enter a ghost process, it must be crossing a host boundary, so the ghost can always respond to *enter* with *suspend*. The location protocol was extended with a *start-new* request that causes a new agent to be spawned with a provided initial state; this is used by *ghost* processes to restart suspended agents, and is also useful for dynamically injecting new agents into a running simulation.

Figure 6 shows a sample process diagram for the refactored distributed simulation; Figure 7 shows the extended protocols, and Figure 8 shows the phases.

We measured the performance of the resulting simulation on a 31-node cluster at Kent with the same network architecture as the Display Wall. We found that the simulation speed was (within a margin of error) independent of the number of hosts in the simulation—as expected, since all the hosts perform their communications in parallel, and only communicate with their immediate neighbours—which demonstrates excellent scalability: the simulation size can be increased by adding more nodes to the simulation. However, the time taken per simulation timestep was significantly larger when running on the cluster than when running on a single host—owing to the latency involved in network communication.

C. Asynchronous Messaging

Since the clusters available to us are built from commodity PC hardware with conventional network interfaces and off-the-shelf operating systems, there is nothing we can reasonably do about the latency inherent in network communication, interrupt handling or operating system processing. On the other hand, we can speed up the simulation by avoiding *round trips* across the network—cases where we send a packet and have to wait for a reply until we can continue, incurring two lots of network transmission latency.

Both occam-π's local channels and pony's network channels implement the full CSP synchronisation semantics: a write will block until there is a corresponding read, and vice versa. This requires a network round trip to acknowledge that the communication has completed at both ends. However, client-server communication usually does not require this synchronisation behaviour, because a request message can be explicitly acknowledged with a response message; asynchronous, buffered messaging suffices for implementing client-server connections. (Similar techniques are common in languages such as Erlang [18] that use asynchronous messaging as a communication mechanism.)

We built Trap, an efficient asynchronous messaging system for occam-π and Python, as part of an investigation into process-oriented implementations of MPI-style collective operations [19]. Trap provides order-preserving asynchronous network channels, implemented using a lightweight TCP-based protocol. In addition, Trap includes an IO scheduling system that allows the occam-π system to efficiently perform a set of communications with the minimal number of operating system calls, further reducing overheads.

As a result of the previous refactoring, the only channels that now need to span the network are the connections between ghosts and their corresponding real locations. This

```
location
  = (enter | move(VECTOR))
      -> (stay-here | go-there(LOCATION)
          | (suspend -> suspended(STATE)))
  | look -> VIEW-LIST
  | start-new(STATE)
viewer
  = look -> VIEW-LIST
agent
  = move(VECTOR)
      -> (stay-here
          | (suspend -> suspended(STATE)))
  | look -> VIEW-LIST
```

Fig. 7. Protocols in the distributed simulation.

1. Ghost processes update
2. Viewer processes update
3. Agent processes retrieve views and perform actions

Fig. 8. Phases in each timestep of the distributed simulation.

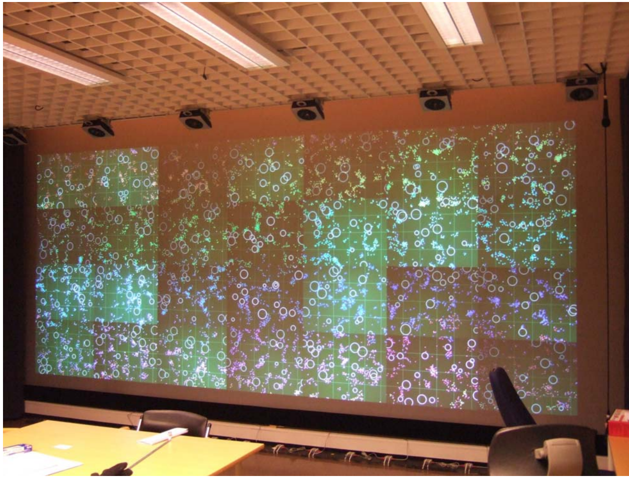

Fig. 9. Occoids running on the Tromsø Display Wall.

made it very straightforward to replace these channels with Trap connections. The resulting simulation has the same excellent scalability as the pony-based simulation, but runs significantly faster, since it only requires half the number of network round trips for each *view* request from a ghost process to its underlying location, all of which occur in parallel in phase 1.

In the case of *start-new* messages sent during agent migration in phase 3, no round trip is necessary at all, as the message requires no response. This works particularly well when a flock of boids migrate together across a host boundary, since their messages will be batched together by Trap for transport across the network.

V. The Display Wall

We have successfully run our distributed version of Occoids on the Tromsø Display Wall, dividing space up among hosts in a way that corresponds to the display tiles on the wall (Figure 9). Each host also runs a visualisation process that is responsible for drawing the contents of its locations. Since the visualisation is performed locally, it involves no network traffic beyond that already necessary to distribute the simulation; unlike other Display Wall applications, no graphics need be sent around.

Visualising our simulation on the Display Wall has a number of advantages. The Display Wall is large enough to completely fill a user's field of vision, making it possible to get an "immersive" overview of the behaviour of a large simulation as a whole, while at the same time having high enough resolution that the details of individual interactions can be easily seen. We believe that an effective visualisation of a complex system can be a powerful tool for helping to understand the behaviours of the system. We found high-resolution visualisation to be very useful when debugging Occoids: the boids should move in a "natural" way when the simulation is running correctly, so many kinds of anomalous behaviours – such as discontinuities at the edges of locations, or errors in the boid movement computations – can be made obvious by the filtering abilities of the human eye.

Synchronisation of display updates is generally a concern when doing distributed visualisation. Our visualisation processes draw one screen update per simulation cycle, during phase 2 of the simulation. While the phases of the whole simulation are implicitly locked together by the communications performed between hosts in each phase, it is still possible for the display updates to occur at slightly different times on different hosts – for example, a host containing many agents will take longer to draw its visualisation than an empty host. In practice, we have found this not to be a problem when visualising Occoids; since the moving boids are small, the unsynchronised display updates are not visually distracting. For visualisation involving moving entities that span multiple tiles – for example, 3D objects – tighter synchronisation would be necessary, and could be obtained using a dedicated display barrier.

One shortcoming of our current implementation is that agents that span multiple hosts – in particular, the trees in Occoids – are only drawn on the host that is running the corresponding agent process. We intend to fix this by making use of the ghost processes to make agents on other hosts available to the local visualisation.

VI. Conclusions

We have shown how process-oriented programming techniques can be used to construct a model of continuous space for complex systems simulations with a high degree of internal parallelism. We have described how such a model can be refactored to run in an efficient and highly-scalable manner upon a cluster of commodity PCs.

While we built the space model initially for use in Occoids, we have since reused it in several other complex systems simulations, including Amos' ant-based annular sorting [20] and a model of lymphocyte migration in high endothelial venules [21], [22]. These new applications required only slight extensions to the space model described above: for the ants model, the location protocol was extended to allow agents to remove other agents from the simulation, and for the lymphocyte migration model, the coordinate system was changed from two-dimensional to three-dimensional space. The techniques described above for distributing Occoids can be applied directly to these new simulations.

We have also performed experiments with Occoids in which we introduced new types of agents such as food and predators, and new behaviours for the boids. These required no modification to the space model at all.

Through our work with the Display Wall, we have demonstrated how distributed simulation can be combined with distributed rendering to provide efficient, high-resolution visualisation of the behaviour of a complex system. We have discussed some of the issues that arose while implementing distributed visualisation, and how we plan to address them.

We plan to experiment further with dynamic load-balancing between hosts in a cluster; this would be particularly useful for a flocking model such as boids in which

the number of boids in a region of space can vary greatly as time progresses. One way to achieve this would be to dynamically vary the sizes of locations so as to expand quiet regions and contract busy regions, migrating agents between them to balance the load.

In addition, we plan to consider potential enhancements to process-oriented languages and libraries to ease the construction of distributed simulations such as these, such as native support for asynchronous network communication and more flexible mobile process facilities. Over the long term, we would like to investigate the possibility of a distributed runtime for process-oriented systems that could perform some degree of automatic load-balancing for a distributed simulation.

ACKNOWLEDGEMENTS

This work is part of the CoSMoS project, funded by EPSRC grants EP/E053505/1 and EP/E049419/1. The work is also partially supported by Norwegian Research Council project NFR 155550/420.

REFERENCES

[1] A. T. Sampson, P. H. Welch, and F. R. M. Barnes, "Lazy Cellular Automata with Communicating Processes," in *Communicating Process Architectures 2005*. IOS Press, 2005, pp. 165–175.

[2] C. G. Ritson and P. H. Welch, "A process-oriented architecture for complex system modelling," in *Communicating Process Architectures 2007*, vol. 65. IOS Press, 2007, pp. 249–266.

[3] C. W. Reynolds, "Flocks, herds, and schools: A distributed behavioral model," in *14th Annual Conference on Computer Graphics and Interactive Technologies (SIGGRAPH87)*. ACM, 1987, pp. 25–34.

[4] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, T. Housel, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. P. Singh, G. Tzanetakis, and J. Zheng, "Building and using a scalable display wall system," *IEEE Computer Graphics and Applications*, vol. 20, no. 4, pp. 29–37, 2000.

[5] D. Stødle, T.-M. S. Hagen, J. M. Bjørndalen, and O. Anshus, "Gesture-based, touch-free multi-user gaming on wall-sized, high-resolution tiled displays," in *Proceedings of the 4th International Symposium on Pervasive Gaming Applications, PerGames 2007. Salzburg, Austria, June 2007*, 2007.

[6] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[7] R. Milner, *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[8] P. H. Welch and F. R. M. Barnes, "Communicating mobile processes: introducing occam-pi," in *25 Years of CSP*, ser. Lecture Notes in Computer Science, A. E. Abdallah, C. B. Jones, and J. W. Sanders, Eds., vol. 3525. Springer Verlag, April 2005, pp. 175–210.

[9] P. H. Welch, "Process Oriented Design for Java: Concurrency for All," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, H. R. Arabnia, Ed., vol. 1, CSREA. CSREA Press, June 2000, pp. 51–57.

[10] C. G. Ritson, A. T. Sampson, and F. R. M. Barnes, "Multicore scheduling for lightweight communicating processes," to appear.

[11] J. M. R. Martin and P. H. Welch, "A Design Strategy for Deadlock-Free Concurrent Systems," *Transputer Communications*, vol. 3, no. 4, 1997.

[12] A. T. Sampson, "Two-Way Protocols for occam-pi," in *Communicating Process Architectures 2008*, ser. Concurrent Systems Engineering, P. H. Welch, S. Stepney, F. A. C. Polack, F. R. M. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson, Eds., vol. 66, WoTUG. Amsterdam, The Netherlands: IOS Press, September 2008, pp. 85–97.

[13] F. R. M. Barnes, P. H. Welch, and A. T. Sampson, "Barrier synchronisation for occam-pi," in *2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. CSREA Press, 2005, pp. 173–179.

[14] P. S. Andrews, A. T. Sampson, J. M. Bjørndalen, S. Stepney, J. Timmis, D. N. Warren, and P. H. Welch, "Investigating patterns for the process-oriented modelling and simulation of space in complex systems," in *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, S. Bullock, J. Noble, R. Watson, and M. A. Bedau, Eds. MIT Press, Cambridge, MA, 2008, pp. 17–24.

[15] M. Schweigler, "A Unified Model for Inter- and Intra-processor Concurrency," Ph.D. dissertation, Computing Laboratory, University of Kent, Canterbury, UK, Aug. 2006.

[16] M. Grand, *Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML*. John Wiley and Sons, 1998.

[17] F. R. M. Barnes and P. H. Welch, "Communicating Mobile Processes," in *Communicating Process Architectures 2004*, ser. Concurrent Systems Engineering Series, I. East, J. Martin, P. H. Welch, D. Duce, and M. Green, Eds., vol. 62. Amsterdam, The Netherlands: IOS Press, September 2004, pp. 201–218.

[18] J. Barklund and R. Virding, "Erlang 4.7.3 Reference Manual," Feb. 1999.

[19] J. M. Bjørndalen and A. T. Sampson, "Process-Oriented Collective Operations," in *Communicating Process Architectures 2008*, ser. Concurrent Systems Engineering, P. H. Welch, S. Stepney, F. A. C. Polack, F. R. M. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson, Eds., vol. 66, WoTUG. Amsterdam, The Netherlands: IOS Press, September 2008, pp. 309–328.

[20] M. Amos and O. Don, "An ant-based algorithm for annular sorting," in *Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC)*. IEEE Press, 2007, pp. 142–148.

[21] J. Girard and T. Springer, "High endothelial venules (HEVs): specialized endothelium for lymphocyte migration," *Immunology Today*, vol. 15, pp. 449–457, 1995.

[22] P. S. Andrews, F. A. C. Polack, A. T. Sampson, J. Timmis, L. Scott, and M. Coles, "Simulating biology: Towards understanding what the simulation shows," in *Proceedings of the 2008 Workshop on Complex Systems Modelling and Simulation, York, UK, September 2008*, S. Stepney, F. Polack, and P. Welch, Eds. Luniver Press, 2008, pp. 93–123.