Technical Report CSTN-052

Simulation Modelling and Visualisation: Toolkits for Building Artificial Worlds

D.P. Playne, A.P. Gerdelan, A. Leist, C.J. Scogings and K.A. Hawick*
Complex Systems & Simulations Group (CSSG)
Institute of Information and Mathematical Sciences
Massey University, North Shore 102-904, Auckland, New Zealand
Email: {daniel.playne, gerdelan}@gmail.com, {a.leist, c.scogings, k.a.hawick}@massey.ac.nz
Tel: +64 9 414 0800   Fax: +64 9 441 8181

August 2008

Abstract

Simulations users at all levels make heavy use of compute resources to drive computational simulations for greatly varying applications areas of research using different simulation paradigms. Simulations are implemented in many software forms, ranging from highly standardised and general models that run in proprietary software packages to *ad hoc* hand-crafted simulations codes for very specific applications. Visualisation of the workings or results of a simulation is another highly valuable capability for simulation developers and practitioners. There are many different software libraries and methods available for creating a visualisation layer for simulations, and it is often a difficult and time-consuming process to assemble a *toolkit* of these libraries and other resources that best suits a particular simulation model. We present here a break-down of the main simulation paradigms, and discuss differing toolkits and approaches that different researchers have taken to tackle coupled simulation and visualisation in each paradigm.

Keywords: simulations; graphics; visualisation; artificial realities; model worlds.

Contents

*Author for Correspondence

1 Introduction

Simulation and modelling continue to play important roles underpinning the computational sciences [1,2]. Simulations come in many different forms and degrees of complexity. These range from simple operational prediction models that might be made using nothing more sophisticated than a spreadsheet, through the use of general purpose modelling packages and environments to the development and use of custom hand-crafted and optimised simulation codes that run on supercomputers and other dedicated hardware systems.

Some simulations applications are very well known such as the problems of weather and climate prediction, both of which make use of many supercomputers around the world. Some are more mundane sounding but of high economic importance such as modelling air-flow and drag across new car and aircraft designs. Other applications are more esoteric and less well known such as the various simulation programs used to simulate the effects and associated phenomena of nuclear explosives.

An important idea that has been progressively up taken over the last fifteen years is Fox's concept of simulation on demand [3], whereby simulation programs are organised as services and can be accessed by client programs or indeed through a web interface. This approach can be used to make custom simulations more accessible to a wider user population. Complex simulation programs that are difficult to use can be wrapped up in a service-oriented software infrastructure such as web forms or even an immersive graphical interface. Another important related idea is Smarr's concept of steering computations [4] using advanced graphics or even totally immersive virtual reality systems. Support for this idea enables simulation users to home in on the parameter region of their problem that is of interest by facilitating a fast and close interaction between user and the simulation running on supercomputing resources.

These ideas can be combined and it is possible to consider how the major classes or paradigms of simulation can make use of these notions and the tools and technologies that are already widely available. Many simulation categories have a strong need for good visualisation capabilities. A simulation is often best debugged (during development) and understood and interpreted (during production use) with the aid of a visual representation of the system configuration.

In this article we review some of the main simulation architectural ideas (Section 2) including: batch-driven simulations (Section 2.1); systems where the visualisation is driven by the simulation algorithm (Section 2.2) and systems where the simulation is driven by some intrinsic agent component (Section 2.3). We also discuss some cross-cutting issues (Section 3) for simulation developers including: accuracy (Section 3.1); repeatability (Section 3.2); validation (Section 3.3); computational performance (Section 3.4); and complexity (Section 3.5).

We believe that most simulations are based around a key idea or paradigm (Section 4) such as particle-based models (Section 4.1); models built around partial differential equations (Section 4.2); or models based on discrete events in time or space (Section 4.3). A complex application may of course be a hybrid of all these (Section 4.4).

Most of the simulations applications of direct interest to ourselves (Section 5) are complex and dynamical systems that might be based on particle collision dynamics (Section 5.1), gravitational particle models (Section 5.2), animat agents (Section 5.3), graph-based networks (Section 5.4), and artificially intelligent robotic agents (Section 5.5). In general these can all be described as **complex systems** with highly non-linear behaviours that can only be adequately explored using a closely coupled combination of simulation and visualisation methods. A key problem is to use the visual representation of the simulated model to explore its parameter space and to try to categorise the phenomena. Ideally the reductionist scientific approach is feasible and a complex system can be described by a general law or analytic descriptive formula. In this sense, these simulation applications are examples of the computational science approach [5] - using numerical experiments to link theoretical ideas with experimental observations.

A number of software tools (Section 6) are available to aid development and operation of simulation applications such as we describe. Some standard software packages (Section 6.1) can be used directly for some simulations, whereas in many cases an appropriate programming language (Section 6.2) must be used to engineer a new simulation code. Such development work often can make use of library software for scientific computations (Section 6.3) and visualisation (Section 6.4).

We offer some observations on how all these ideas on simulation and visualisation are typically taught to students in Section 7 and summarise our conclusions and ideas for the future in Section 8.

2 Software Architectures

When implementing a simulation and corresponding visualisation engine, correct software design is vital to the re-usability and portability of the system. A simulation that is designed in an object-oriented and modular way can allow many components to be reused with little or no modification. Some simulations require a high-degree of integration between the simulation and the visualisation engine which limits the modularity of the software. However, some simulations can be almost completely separated from the visualisation and can communicate purely via an interface. Presented here are three common simulation structures that deal with these types of simulation: Batch-Driven architecture; Simulation-Driven architecture; and Agent-Driven architecture.

2.1 Batch-Driven Architecture

The **Batch-Driven** architecture is the simplest architecture to implement as it does not involve real-time or interactive visualisation. Multiple instances of the simulation are executed to provide a set of results about the simulation for certain parameters or parameter ranges. These calculated results can be then analysed or final states turned into a single visualisation. Figure 1 shows the basic architecture of a Batch-Driven simulation, in whichthe analysis and visualisation are performed separately from the model.

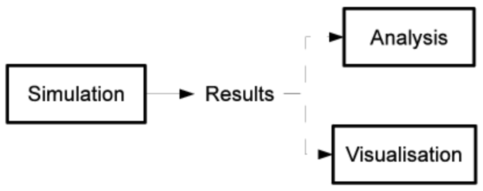

Figure 1: Batch-Driven Architecture.

Simulations designed around the Batch-Driven Architecture are most commonly intended for gathering statistical data about a well developed simulation model. Each simulation instance is examined for some interesting phenomena or some property of each simulation is measured to provide this statistical data. Statistical gathering batches are often used to prove theories about simulations with certain configurations.

Batch-Driven simulations are also used to search a simulations parameter space. Batches of multiple simulation instances are executed with different parameters to search for a specific or many phenomena. These batches can be used to show or discover what conditions are necessary within the simulation for some event or phenomena to occur.

Two common scenarios are when a series of batch driven simulation runs are needed to explore the parameter space of a model or to explore the sensitivity (or lack thereof) to different starting conditions of a model system. In the former case managing the partial cross product of all the possible parameter values is important [6]; in the second case the key issue is managing an appropriate set of (pseudo-)randomly [7] generated configurations and the corresponding simulation runs.

2.2 Simulation-Driven Architecture

A **Simulation-Driven** architecture system is driven by the simulation. The entities within the simulation interact over time and are displayed asynchronously by the visualisation engine. The simulation is completely unaware that the visualisation engine exists and the entities will interact the same regardless of whether the simulation is being visualised or not. The simulation will provide a list of the entities within the simulation, and this list can then be visualised.

The visualisation engine is responsible for displaying a list of entities and handling user input to change the method in which the entities are displayed. The visualisation engine will handle user input which controls only the visualisation. This input can be in the form of options displaying extra information about a simulation such as: grids, trails, and energy values.

One very common type of visualisation control is navigating a three dimensional scene. If the simulation is in three dimensions then it is often impossible to see the entire simulation from one view point. In this case the user can control the visualisation and change the perspective of view of the simulation to see different parts of the system.

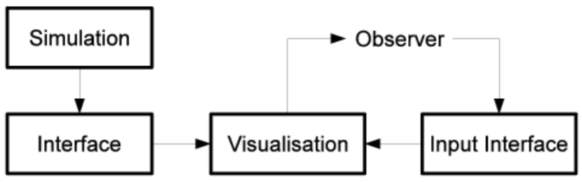

Figure 2: Simulation-Driven Architecture.

The actual communication between the visualisation engine and the simulation is performed by an interface object. This object is created with specific knowledge of both the visualisation engine and the simulation and must be able to communicate with the simulation, retrieve the required entities and send them to the visualisation engine for processing. A new interface must be created for each simulation and visualisation engine. The size of the interface is minimal so implementing a new interface for each simulation requires a small amount of effort. Figure 2 is a diagram of the simulation driven architecture.

In this architecture the operation of the simulation is the point of interest. The final result is defined by the interactions between the entities within it based on their starting configuration and other parameters of the simulation. This result may be in the form of a forecast where the simulation predicts the state of a system in the future given a starting configuration. Another possibility is that the visualisation of the simulation is examined by an observer to identify interesting events or phenomena inherent to the system. These simulators model systems such as Particles (Section 4.1), Field Equations (Section 4.2) and Animats (Section 5.3).

2.3 Agent-Driven Architecture

In the **Agent-Driven** architecture the simulation itself serves a very different purpose, it provides a test-bed environment for intelligent agents. The simulation defines a set of rules and parameters about how the entities within it may act but allows the entities to change their state. What actions the entities perform to change their state is decided by an outside controlling agent. In this architecture the result of the simulation is dependent on the decisions made by the controlling agents.

These agents have a degree of control over one or more entities within the simulation and control them according to decisions they make based on the state of the simulation. The decisions made by the agents control how the entities act within the environment they exist in and how they interact with the other entities. These agents can be controlling human operators or artificial intelligence control programs. In Figure 3 the controlling agents are shown distinct from the simulation as they are most commonly separate programs that operate outside the simulation.

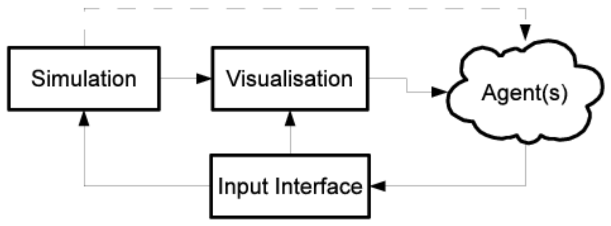

Figure 3: Agent-Driven Architecture.

When a human operator is the controlling agent then the simulation is no longer merely visualised by the graphics engine, it is inextricably linked to it. The human operator processes the visualisation to determine the state of the simulation and from this information makes decisions about how to control its entities. This decision is then input into the simulation via an input interface (see Figure 3). Now the user input handled by the visualiser not only controls the method in which the simulation is visualised but also feeds back to control entities within the simulation. Because of this the simulation can no longer operate irrespective of the visualisation because it must receive input from the operator via the input interface.

When the controlling agents are artificial intelligence programs rather than human operators then the input into the simulation comes from an interface to the agent program. The information about the state of the simulation is sent to the agent directly without the need for the visualisation engine. The agent program can base its decision based on the information it receives from the simulation and then makes its decision via the input interface.

Although the visualisation engine is not directly utilised by the agents within this system, it is still vital for a human operator to observe the simulation and examine the actions of the agent programs to ensure they are working as intended. In this case the human operator will still have control over the visualisation engine with the exception that the input is no longer feeding into the simulation, similar to the Simulation-Driven Architecture the user input will simply control the method in which the simulation is visualised.

The Agent-Driven architecture does not focus on the operation of the simulation but rather on the decisions made by the controlling agents. The simulation merely acts as a restricted testing environment to examine the performance of the agents. Each agent has a goal that it will try to achieve by controlling its entities within the limits of the simulation. Applications utilising this architecture include: most modern computer games, robot soccer simulators and robotic agent simulators (Section 5.5).

3 Simulation Considerations

There are a number of important cross-cutting issues that affect all of the simulation paradigms that we have discussed. These include: accuracy and precision; repeatability; validation and verification; performance; and complexity.

3.1 Simulation Accuracy and Precision

Numerical accuracy can vary from vitally important to relatively insignificant. The importance of accuracy depends on the workings and the purpose of the simulation. Accuracy is often most important for simulations built on the Simulation-Driven Architecture. In this architecture the operation of the simulation is the focus, if this simulation is inaccurate then the results will be inaccurate and unreliable.

Simulations designed around the Agent-Driven Architecture are often not required to be as accurate. Because these simulators are only providing an environment to test the controlling intelligent agents, the actual operation of the simulation is less important. However this is not always the case. In some simulators such as robot soccer systems (see Section 5.5, the models are designed to simulate a real-world environment where physical realism is important.

In this case the agents' properties can only be tested by a physically realistic simulation, thus the level of accuracy in these simulators will enhance the value to real-world application of any algorithms developed to operate within the simulated environment.

3.2 Simulation Repeatability

Repeatability is highly important for simulation and especially for those designed around the Simulation-Driven Architecture. Two executions of the same simulation that are initialised with the same configuration should both compute the same final state or result. This repeatability becomes increasing hard to ensure when the simulation incorporates a degree of randomness within its computation. Simulations that incorporate randomness within experiments require a method of managing and repeating it. There is little point in discovering an interesting effect or phenomena if the experiment cannot be repeated.

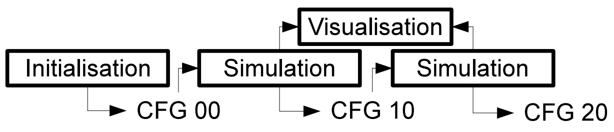

Figure 4: Configuration-Chaining Architecture.

Figure: 4 shows a way of managing random-number generator (RNG) configurations. The simulator loads a configuration file which stores a starting configuration for the system as well as the RNG. The simulation then computes the interactions of the entities within the system over a period of time and saves a final configuration. The configuration of both the simulation and the RNG state is saved and can be reloaded at any time. It is important to ensure that executing the simulation for ten time-steps, twice should produce the same final configuration as one execution for twenty time-steps. This method of configuration managements allows the simulation configuration to be reloaded at the start or end of any execution.

The configuration of the visualisation is not saved within the configuration file. The view of the system need not be saved as the user can easily configure the visualisation to any setting they desire at any time. As long as the simulation itself can be repeated then the user can easily configure the visualisation engine to show the same view or a new view of the same event. This method allows the user to watch the same event repeatedly from different angles of view to provide a better image of the workings of the system.

3.3 Validation and Verification

Computer simulation is often used when testing the hypothesis that observed behaviour from a complex system truly emerges from a simple guessed microscopic model for the system's constituent parts. There may be strong microscopic physics arguments behind the choice of the component model or it may simply be a plausible guess. In these experiments it is important to conduct enough simulation runs over a properly varying set of choices of microscopic rules before it can be reasonably concluded that the model is consistent with the observations. One surprising feature of complex systems is the emergent universality of some sorts of complex behaviour independent of a wide variety of microscopic model parameters.

In some cases specific observations may be available for example of a physical system or a modelled crowd or a simulated network with which the simulation can be compared. This is not always the case however and a common driving force to use simulation is to help make a computational science link between the two conventional approaches of theoretical analysis and experimental observation. A well-posed simulation using well understood microscopic components can aid considerably in understanding the complex emergent macroscopic behaviour of a whole system.

3.4 Performance

The performance of a computational simulation becomes increasingly important as the size of the simulation increases. Simulations must complete in a reasonable length of time for their results to be useful. In many cases this requires simulations to be structured differently in order to operate on supercomputers, computational grids, special hardware such as Graphical Processing Units (GPUs) or even Field-Programmable Gate Arrays (FPGAs) or other Application-Specific Integrated Circuits (ASICs).

This restructuring of the software often means that the architecture best suited to the simulation must be modified and in some cases broken. In many cases there is no way to avoid this situation, sometimes clean architecture design must be sacrificed for the necessity of performance.

As Knuth warns, premature optimisation is undesirable [8]. This should not be taken to mean that subsequent optimisation is unimportant. It is. Many simulations projects would be completely infeasible without careful optimisation. A classic case is the work reported in [9] which took many CPU-years even with assembler-level optimisation.

3.5 Complexity

It is increasingly the case that we are targeting simulations of complex systems that may involve a hybrid of the techniques discussed in this article. This inevitably leads to a more complicated simulation architecture.

An overly complicated simulation inevitably raises concerns about validation and verification and also reproducibility, and often performance. A clean simulation software architecture that can be validated as individual components helps allay these concerns.

Reuse-ability - which might deserve to be a criteria in its own right - is closely tied to complexity and performance. Generally we hope to have very modular software that can be unit tested and verified. This additional effort is amortized over greater use if the module is widely applicable. Unfortunately achieving performance often involves compromises and tradeoffs that reduce reuse and raise code complexity.

4 Simulation Paradigms

There is considerable variety in the paradigms of simulations used for scientific research. The method of visualisation for each of these simulation paradigms is equally variable. This section presents a number of simulation paradigms and discusses the methods and libraries used to visualise them.

4.1 Particle-Based Models

Particle dynamics simulations model the motion and behaviour of particles or objects in space. A particle is considered to be a single point mass in space with a position and velocity [10,11]. A particle's motion is normally constrained by Newton's Laws of Motion [12] and by some potential equation or some external force acting upon all particles. The potential equation of a simulation describes an attractive/repulsive force between a pair or particles. In each simulation, particles have a set of properties such as radius, charge or spin that also define their state in addition to their position, velocity and mass.

These common properties create the opportunity to graphically display the particles simply as an object in space according to its size, shape and position in the system. The particles' motion over time is rendered as video with each frame showing the particles at different positions. This visualisation can be done in real-time as the simulation runs with lower quality, or be rendered as a high-quality movie recreated from recorded positions after the simulation has finished. These particle simulations can be run in any number of dimensions but once again our visualisation of them is limited to the three-dimensions we have the capability to display.

Like the field equation simulations (see Section 4.2), particle simulations are best suited to the Simulation-Driven Architecture. The focus and interest of the simulation is the behaviour and interaction of the particles. Within a particle simulation the two main issues are: their relative motion and attraction due to the potential between them; and their behaviour when the particles collide. This is all focused on the simulation, the visualisation engine's purpose is to simply display the particles.

We discuss the application of the particle model to the problem of particle collision fluid modelling in Section 5.1 and gravitational astro-dynamics in Section 5.2, but this paradigm is also useful for modelling atomic or molecular systems. Totally unphysical "pseudo particles" can also be used to assist in rendering materials such as flowing water or the moving cloth. Pseudo particles can form a tethered system of anchor points to form a dynamic surface upon which a texture map can be draped for rendering.

4.2 Field Equation Systems

Field equations simulators model the behaviour of microscopic atoms over a discrete macroscopic cell. The modelled system is split into discrete cells and the state of each cell is defined by a set of properties. Every cell interacts with the cells in the area or volume surrounding it and changes its state according to the properties of the surrounding cells and field equations of the simulation. Visualising these simulations involves displaying some property or combination of properties that define each cell in a graphical way. This is often performed by displaying each cell at its appropriate position and with a colour defined by the properties of the cell.

These field equation simulations can be created for any number of dimensions, however our visualisation techniques are currently limited to three. Each dimension involves displaying the cells in a different arrangement, in general – one-dimensional simulations are a line of cells, two dimensions require a plane, and three dimensions are represented by a cube of cells.

Visualising these cells in one and two dimensions is relatively easy as a simple 2D graphics package can be used and a view of the entire simulation can be easily seen. The visualisation of a cube of cells requires more careful consideration. A 3D graphics library must be used to visualise the system and should provide the user with a way of changing the view of the scene but also provide a method of seeing inside the cube. If all the cells of the system are entirely opaque then only the outside layer of cells can be seen at any point in time.

Figure 5 shows the visualisation of a three dimensional field equation. This cube of cells has been displayed based on a concentration of A and B atoms. If a cell contains mostly A atoms then it is coloured blue and is entirely opaque. However if the cell contains mostly B atoms then it is coloured yellow and is displayed as almostly entirely transparent. This allows the state of the cells at the centre of the cube to be seen from the outside. This is one method of displaying a cube so the state of all cells can be seen.

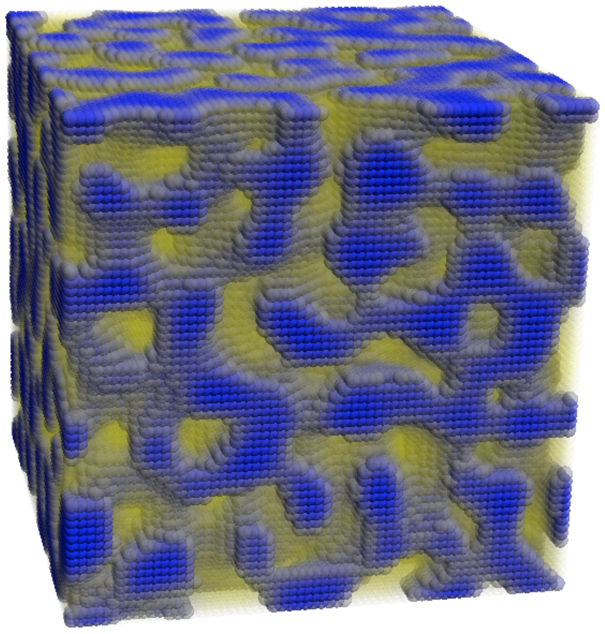

Figure 5: A three-dimensional visualisation of the Cahn-Hilliard-Cook field equation.

Figure 5 is in fact a visualisation of a three-dimensional Cahn-Hilliard-Cook field equation simulation. The Cahn-Hilliard Cook equation models phase separation in a binary fluid. Started from a random uniform configuration, surface tension effects drive the field cells to gradually coalesce and merge into separate domains. This domain separation forms the emergent camouflage pattern seen in the visualisation. This equation can be used to model the phase separation in cooling alloys to discover the optimum cooling process for forming real-world metals [13].

The result of the model for a given starting configuration is the focus of the simulation. A study is made of the different exhibited behaviours of the model, that are dependant upon control parameters such as temperature for example. Thus these simulations are best suited for the Simulation-Driven Architecture. The simulation is developing independently from the visualisation engine and is unaware of its existence. The visualisation engine used to create figure 5 was created using the JOGL [14] graphics library.

The visualisation engine uses the Java interfaces to handle the user input and uses the JOGL functions to change the view in a 'fly through' camera style. This allows the user to navigate through the scene using the mouse and keyboard. As the JOGL engine is designed as an interface between Java and OpenGL, it is obviously best suited to operating with simulations implemented in Java. However, it would be a relatively simple process to convert this JOGL visualisation into a C++ OpenGL engine.

4.3 Event-Driven Systems

The event driven paradigm is commonly used in cases where some aggregate or emergent behaviour arises from the collective interactions of many discrete participants. Events might be arrivals of data packets in a network, or transactions in a management situation, or encounters and conflicts in a defence scenario.

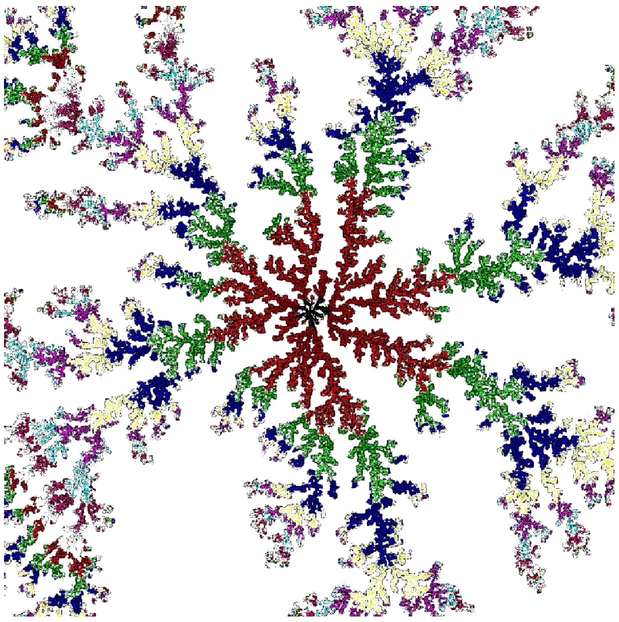

Figure 6: An aggregate on a discrete 2D lattice, grown from a central seed particle by diffusion-limited aggregation.

One example application is that of diffusion-limited aggregation (DLA) on a discrete lattice. Figure 6 shows a fractal cluster grown using a diffusion-limited aggregation algorithm [15]. This is a form of event driven simulation, where a seed cell planted at the centre of the image is gradually grown by releasing "walker" particles around it. The walkers diffuse randomly and attach themselves to the growing aggregate. This cluster was grown on a square mesh by walking and attachment events but it is also possible to employ this algorithm on continuous x-y coordinates. This aggregate has an approximate fractal dimension of d=*1.6*, roughly intermediate between 1 and 2 spatial dimensions. It is also possible to grow aggregates embedded in a 3-D space.

Many physical system can be modelled by the discrete movements of atoms or cells in a system. One famous discrete event model is Conway's game of life [16], which is a cellular automaton. Each spatial cell has very simple microscopic rules governing its temporal behaviour but some very complex and unexpected patterns emerge from the overall collective. Monte Carlo lattice models also fit into this paradigm. Models such as the Ising model [9], Potts model [17], Heisenberg model and clock models are essentially microscopic automata that interact with their local neighbouring cell and through a stochastic dynamical scheme or pseudo time imposed upon them they give rise to complex phenomena such as phase transitions [18].

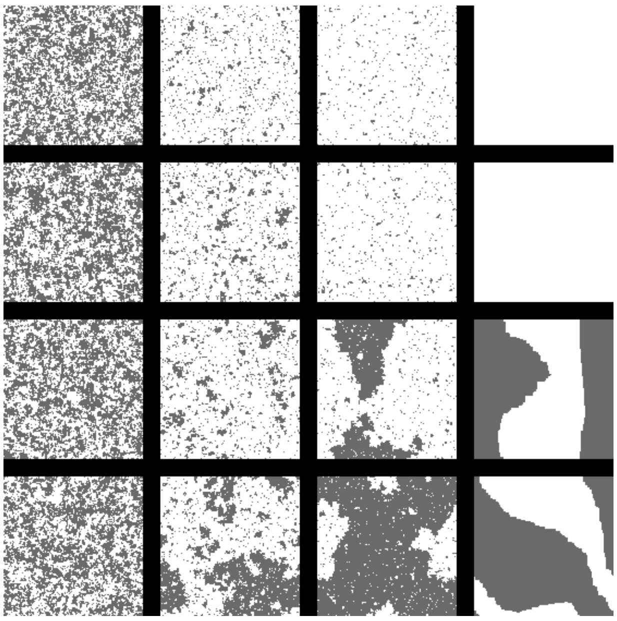
Figure 7: Configurations of a lattice Ising Model of magnetic spins at different temperatures (vertically) and long-range connection probabilities (horizontally)

Figure 7 shows some snapshots of the Ising model on a 128×128 site lattice at different temperatures and long-range connection probabilities. The phase transitions result in very different cluster shapes and resulting magnetic behaviours.

Many systems that have a continuous time scale can also be modelled by specific events that occur at arbitrary times and which can be modelled through queues. These problems are notoriously difficult to fully distribute or parallelise although a distributed simulation can manage separate clocks or time queues that can be rolled back or time-warped to obtain synchronicity across all participating computers [19].

4.4 Hybrid

A particular simulation to model a coupled set of phenomena may need to make use of a hybrid approach. In some cases a hybrid model might link together multiple components of an overall model, each of which fit the same basic paradigm. An example of this would be simulations of the global weather or climate [20]. Typically models run by national bodies such as the UK Meteorological Office will comprise separate field-based models for the atmosphere and the ocean. These may be completely separate codes that exchange data or they may be a single integrated simulation program. Simulation variables such as temperature and pressure fields are separately integrated in the ocean and atmosphere - likely using different model meshes and resolutions, but are coupled together to ensure the correct physical boundary conditions are available to the separate model components.

In the case of a weather simulation, real-time events such as rain, sunshine, and other measured observations are also incorporated into the running simulation as they become available. Operational weather prediction codes are typically based around this hybrid structure [21].

Other predictive simulations such as those used to predict extractive yields in oil reservoir systems are typically also a hybrid of a field model and discrete event information [22]. Particle methods have widespread uses for straightforward systems such as planetary dynamics or molecular dynamics. For some systems such as simulating very low density fluid flow around orbiter spacecraft re-enty for example, a hybrid model of particles and field equations has to be employed [23].

A highly interactive simulation could be constructed where the human user agents are modelled as a stream of discrete events. The hybrid paradigm is therefore quite common in many practical complex simulation applications.

5 Simulation Applications

There are many simulation applications that make use of the paradigms and ideas we discuss in this article. In the following sections we focus on applications of: particle collision dynamics; planetary dynamics; animat agents; complex networks; and robotic agents. Some of these map neatly to a single paradigm as discussed above, however some involve a hybrid architectural approach.

5.1 Particle Collision Dynamics

Particle simulations that incorporate a collision dynamics model can be used to simulate the microscopic interactions within many complex systems. Particles within such systems are no longer considered to be simple point particles but also define an area or volume. This particles are generally modelled as hard spheres with a variety of collision models. Systems that can be simulated by a large number of hard sphere particles include: aerodynamics, liquids, sand and metal alloys [24].

Lennard-Jones Potential

One method of simulating hard-sphere collisions is through the definition of the potential. By specifying potential equation that repulses two particles if they are too close together has the effect of making the particles reflect off each other. One potential commonly used for this purpose is the Lennard-Jones potential [10, 25]. This is defined by the formula:

$$V(r) = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}] \tag{1}$$

where $V(r)$ is the potential energy between two particles as it varies with radial distance r.

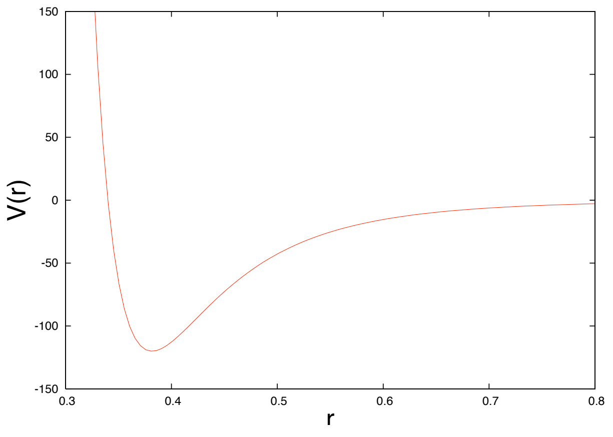

Figure 8: The Lennard-Jones potential.

Figure 8 shows that when the particles come within a certain distance of each other, the potential becomes repulsive and they reflect off each other. This method of collision handling does not model particles as perfectly hard spheres that collide but rather as point particles that repulse each other when in close proximity. This is a very simple method of simulating particles collisions but does not provide an instantaneous perfect reflection at the point the two particles collide.

Reflection

The reflection model only affects particles if they have collided. At the end of each time-step the model searches for collisions between two particles by testing if their volumes overlap [26]. If a collision is detected, the convergent aspect of the particles' velocities are reversed so the particles reflect perfectly off each other. As the simulations compute the particles' motion over a discrete time-step, there will be an error in the collision detection as the particles can actually move into each other, thus causing the overlap. This is physically unrealistic and causes a certain degree of error, however with a reasonable time-step this error can be reduced to an insignificant level. A more physically accurate but computationally expensive collision model is the reflection step-back model.

Reflection Step-Back

The reflection step-back model is designed as a more physically accurate version of the reflection model. When two particles are found to be overlapping, the exact point in time at which the particles' collided is computed and the two particles are moved backwards to this time [26]. The convergent velocities of these particles are then computed and reversed and the two particles moved forward to be in time with the rest of the system (see Figure 9).

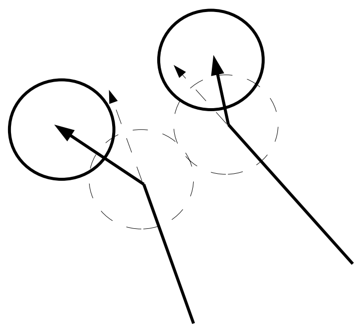

Figure 9: A diagram of two particles colliding with the Reflection Step-Back model.

This is a very accurate and feasible model in the dilute case, however it becomes computationally expensive when there are a large number of particles in close proximity. When there are many closely packed particles, each collision often results in another collision and so on. The large number of time step-backs required to compute the final positions of the particles can cause the simulation to slow down dramatically. The reflection step-back model is very physically accurate but under certain conditions can become too computationally expensive to complete in a reasonable length of time. Another problem that affects both this model and the reflection model is that they may miss a collision. It would be possible for two very fast moving particles to move through each other in a single time-step and if they are not overlapping at the end of the time-step no collision will be detected [26].

5.2 Planetary Dynamics

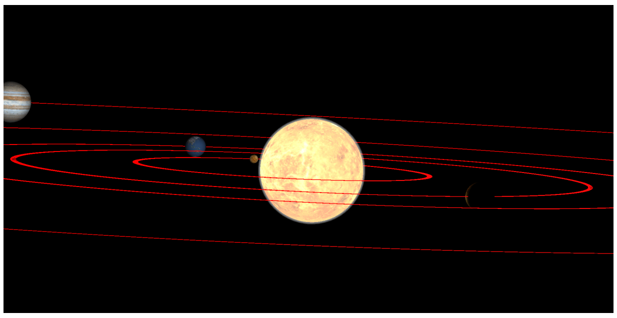

Figure 10: A visualisation taken from a particle system that simulates the Solar System. Image created using the JOGL Z-Buffer engine [14].

Figure 10 shows the visualisation of a three-dimensional particle simulation. This simulation models the specific case of planetary interaction. It computes the motion of planets which are attracted together by the force gravity. These planets are visualised as spheres with textures applied to their surface to make them look more realistic and individually recognisable. The planets in Figure 10 should be easily recognisable as the simulation and visualisation engine used to create them was configured to simulate our solar system. The visualisation has the added detail of displaying trails behind the planets as they move to convey more information to the observer by making it easier to see the path of the planets. The details of this model and some of its implications are discussed in detail in [27].

The actual visualisation engine used to create this image has been implemented in JOGL and is almost identical to the one used by the Cahn-Hilliard-Cook field equation simulation (see Section 4.2). Both these simulations are designed using the Simulation-Driven Architecture and so the visualisation engine is completely separate from the simulation. Thus the same visualisation engine can be used to display both models with little modification.

5.3 Animats

Animat or spatial agent models are widely used in studying artificial life but also for studying sociological as well as physical systems. Animat models such as Tierra [28] and Avida [29] have become widely used tools for studying emergent behaviour in artificial life systems. We have developed our own spatial agent model based on artificially intelligent predator and prey agents that coexist and interact together in a two-dimensional simulated world. There are two unusual features of our animat model. The system has a large number of very simple microscopic agents - typically systems of up to $\approx 10^6$ animats are computationally feasible. The model has a very definite spatial coordinate space and animats interact with other animats within a distance of up to around fifty grid cells. We have discovered a number of emergent or collective phenomena from this model, including boom-bust predator prey population dynamics and associated spatial fluctuations [30]. An interesting discovery was the emergence of spiral wave group fronts arising from the predator and prey interactions [31].

To make effective use of this simulation model we had to develop not only a very efficient simulation program [32] that makes use of multi-phase synchronisation techniques [33], but also a number of metric and quantitative analysis techniques such as semi-automatic cluster classification [34].

Although some attempts have been made to use general purpose simulation packages and languages, for the most part models such as our animat systems and those of other researchers must rely on custom simulation codes. Visualising the results can sometimes be an integrable part of such codes, but to date most work has relied on the use of static configuration dumps that can be rendered as static images, or possibly made into an animation movie sequence by simply exporting many static images in time.

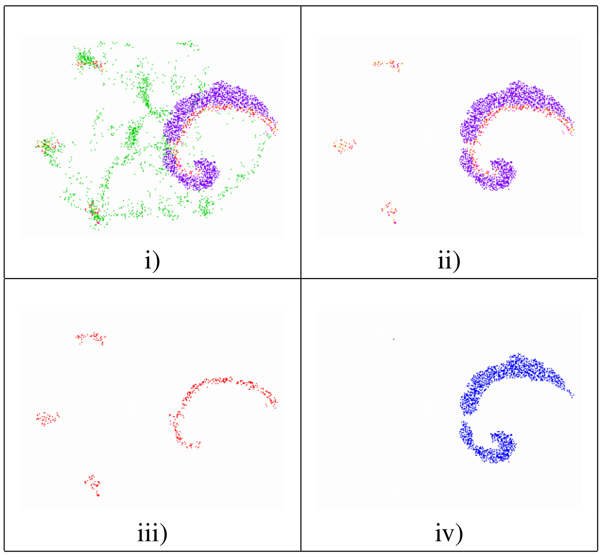

Figure 11: A separation of animat components: i) is the original; ii) the system is shown without the predator corpses; iii) only the live predators are shown; iv) only the live prey are shown.

Figure 11 is taken from [31] and shows some of the emergent spatial structures when predator and prey systems interact. The spiral battlefront is particularly pronounced. Studying animations of the formation and dissolution of these spirals was particularly helpful in understanding the animat collective phenomena.

Our model has had some other surprising uses, such as for modelling resource scarcity effects in ecological systems [35] and for simulating a computational grid - or network of compute resources [36]. Simulating networks in a more general sense is another important application area.

5.4 Simulating Networks

Many interesting physical, technological and social systems can be characterised as a network or graph. While some networks are small enough that they can be directly visualized and various direct counting methods can be used to study their properties, many systems are too big for such easy assimilation. It is however possible to develop a number of useful metrics or quantifiable characteristics that can be used to categorize network systems. Some examples include simple ones such as ratios like edges per node, or fraction of nodes in a a single component; or relative number of components in the full graph. Other more complex characteristics can be derived from colouring the graph to identify its particular components; analysing the pathways through the graph to determine mean minimum and maximum values, or analysing the distribution of loops or circuits in the graph.

Many models can be studied in terms of undirected graphs where we consider nodes and edges, and treat a link from node A to node B as the same as one from B to A. However, certain biological systems and other problems require the use of directed graphs, for which the node reachability can vary depending upon the starting point, even for a fully connected graph. In such cases the pattern or circuits or loops tends also to be more complex.

While some data sets can be studied directly as individual graphs of importance in their own right, a useful approach to studying the properties of networks more generally is to consider graph models that are generated using pseudo-random numbers to draw many samples from a generating formulae. One then samples some metric over many different network samples and attains insights into the general characteristics of the class of model. Investigating properties such as the size of the graph can give insights into the scaling properties and may allow deduction of some limiting behaviour for very large networks that would be infeasible to measure directly.

The simplest case is that of random graphs [37], where each of N nodes might connect to every other node with some probability p and the distribution of edges per node is therefore Poissonian. We can readily generate many such sample random graphs that are all different but are in some sense representative of their class and are characterised by N, p. The notion of simulating graph models thus involves generating many such particular graphs and measuring properties of each one, and studying what properties tend to a sensible characteristic average that is representative of the class and which can ideally be linked through some theoretical model back to the generating or characteristic parameters.

Many such graph or network models exist and we describe some in [38]. One particularly noteworthy general class that has attracted much attention in the literature recently is that of small-world graphs or network systems.

Small-World Networks

Networks are said to show the small-world effect if the mean geodesic distance (i.e. the shortest path between any two vertices) scales logarithmically or slower with the network size for fixed degree k [39]. They are highly clustered, like a regular graph, yet with small characteristic path length, like a random graph [40]. What attracted the attention of scientists from many different disciplines and caused them to investigate such networks is that they can not only be found in books about graph theory, but in social [41–43] and biological [44–46] networks as well.

The small diameter of small-world networks also makes them interesting for computer scientists. Routing algorithms for large-scale, distributed computer networks (e.g. peer-to-peer networks) try to minimise the number of hops needed to send a message from one node to any other node in the network, while also making the networks robust against random failures or deliberate attacks. At the same time they attempt to keep the number of neighbours (i.e. the routing table size) of each node small to reduce the synchronisation overhead of the network. A computer network that possesses the characteristics of a small-world network might be able to fulfill these requirements. One of the main challenges in the design of such a routing algorithm is that it has to be able to find the shortest path—or at least a short path—between two nodes without the knowledge of the entire network structure.

The graphs illustrated in figures 12 and 14 are described in the following section and show, depending on the chosen parameter values, small-world characteristics.

Network Visualisation

Network simulations often use the batch-driven approach (Section 2.1), in which multiple realisations of the same network model are generated for each set of parameters. The metrics of interest are then extracted from each of the instances and averaged to generate charts that can be analysed more easily. To make the results even more meaningful, it is good practice to also calculate the standard deviations from the mean values and to visualise them in form of error bars on the chart.

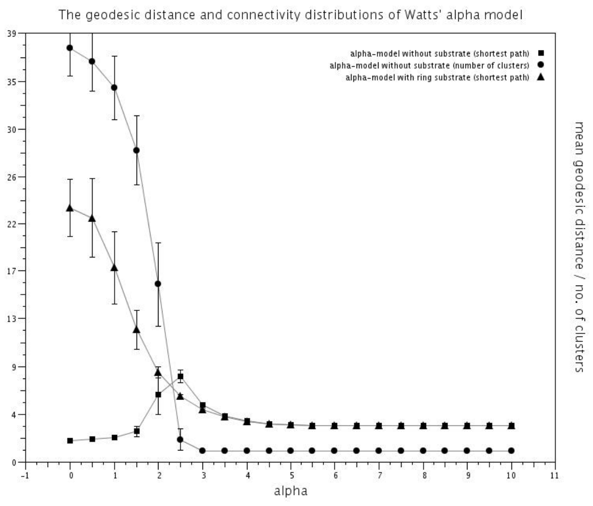

Figure 12: Visualising network metrics in form of charts using Cartan [47]. This chart shows the mean geodesic distance and number of clusters of Watts' α-model averaged over 100 realisations of the graph each. The ring substrate guarantees that the network will be fully connected and, therefore, the number of clusters is always 1 in this case. The $alpha$-values influence the clustering behaviour of the model and range from 0.0 (high clustering) to 10.0 (largely random). The network size $N = 1000$ and the average degree $k = 10$.

An example of such a chart is shown in figure 12. It illustrates how the mean geodesic distance and the number of clusters are related in Watts' α-model [48]. The average shortest path length is small for small α if no substrate is used, but the network is not fully connected. Thus, the distance could also be said to be infinite. The geodesic distance peaks when the different clusters merge together for α-values between 2.5 and 3.0 and then shrinks again when the network becomes more and more random with increasing α. If, on the other hand, a ring substrate is used, then the network is always fully connected. In this case, the geodesic distance is large for small α, because the probability to create random long distance links is very small. It can also be seen that the substrate does not affect the results for $\alpha \geq\approx 4$.

To get an impression of the network and better understand its properties, it is often a good idea to look at a few instances graphically. Which tool is best suited for this task depends on the graph structure. For some graphs it is sufficient to represent them in 2-dimensions, whereas other networks, for instance cubic graph structures or lattices with connected borders like the one illustrated in figure 13, are better represented in 3-dimensions.

The tool used to visualise this graph is UbiGraph [49]. It consists of two separate components. The actual rendering is done by the server, which is currently only available

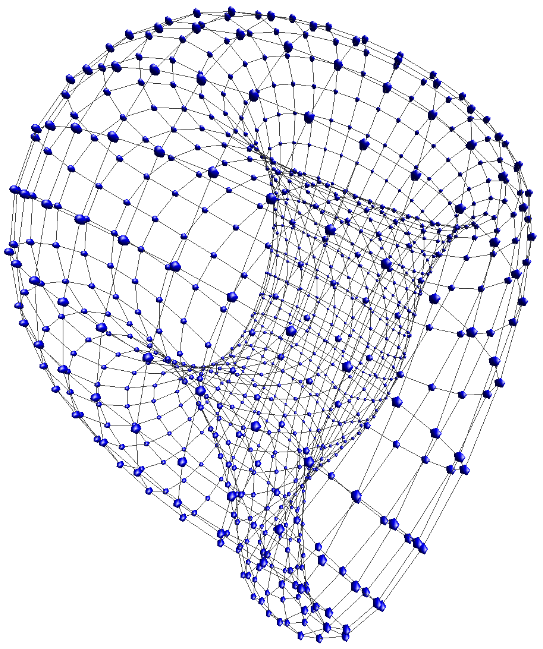

Figure 13: A regular 30×30 lattice visualised with UbiGraph.

as compiled binaries for Linux, Mac OS X and Windows. It uses OpenGL and can utilise up to 8 cpu cores. The client bindings, which use XMLRPC, are available for a number of languages including Java, C(++), Python and Ruby and are distributed under an open source license. UbiGraph supports the dynamic visualisation of graphs through XMLRPC API calls. It enables the manipulation of a number of graph properties, such as the size/width or colour of individual vertices and edges, which can be useful to highlight certain parts of the graph.

While the force-directed layout algorithm used by UbiGraph often produces good results, it is not always what the user wants. Some networks have a regular underlying structure—like a ring, lattice or cube—with only a few additional or rewired "long-distance" links connecting vertices that are not directly adjacent to each other. The largely regular structure of the network quickly gets lost with this layout algorithm, even though the user may prefer to retain it. Due to the closed nature of the UbiGraph server component, it is not possible for us to add further layout algorithms.

GraViz [50] is a graph tool that provides some of these features as illustrated in figure 14. It enables the user to manipulate the graph and supports the automatic layout of rings and chains. The image shown is of a ring of $N = 50$ vertices, initially configured with average connectivity of $k = 2$ but with vertices randomly rewired with probability of $p = 0.2$. The originally connected ring is now fragmented into three separate clusters and the path between vertices 37 and 8 as calculated by the program dynam-

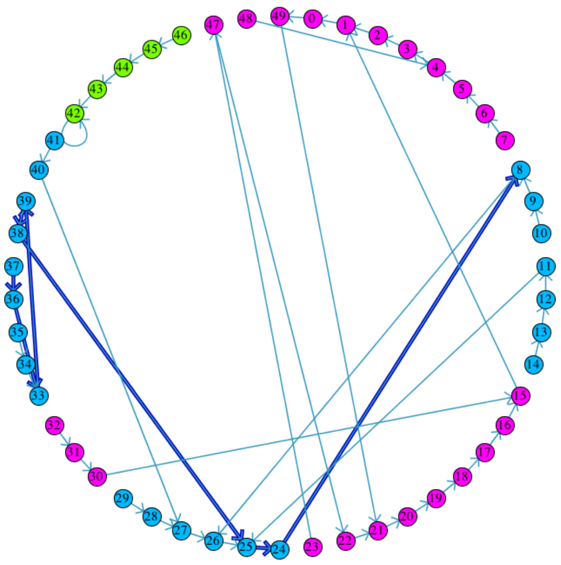

Figure 14: Using GraViz to draw a graph with $N = 50$ vertices and an average degree $k = 2$. The graph started of as a ring and then had its edges randomly rewired with the probability $p = 0.2\%$ [40].

ically, is no longer a simple one. GraViz visualisation is only 2-dimensional and it does not scale very well to large graphs of more than about one thousand nodes. It does also not provide an API to dynamically modify the graph from another program. Other widely available tools for manipulating graphs and graph oriented diagrams include yEd (see [51]) and xfig (see [52]). The former supports layout and manipulation of graphs and can read various graph format files. The latter is a commonly used drawing tool that uses a markup-based file format and which can also be used for drawing or rendering graphs. Unlike GraViz however, these tools do not incorporate path and analysis algorithmic capabilities.

Although the batch-driven approach is often appropriate for network simulations, observing the growth of the network visually can sometimes give additional insight into the network structure or make it easier to understand certain properties of the final network. Thus, the simulation-driven approach (Section 2.2) can also be useful for network simulations.

5.5 Robotic Agent Simulations

Complex simulations of real-world environments require a combination of event-driven, particle, and field visualisation models in order to provide a realistic visual representation. Whilst we have used particle visualisation to approximate more subtle real-world phenomena such as smoke and dust, as seen in Figure 15, terrain and land-

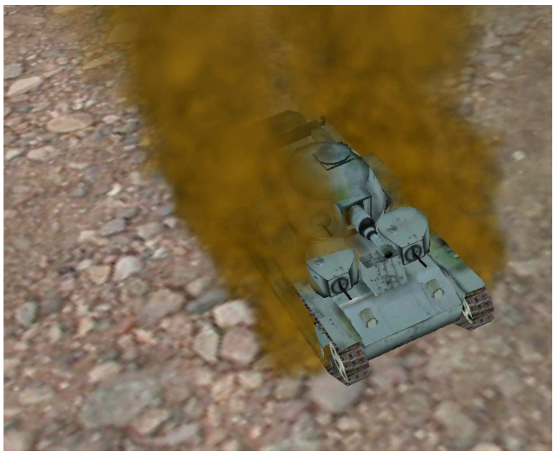

Figure 15: Real-world simulations often combine different visualisation models.

scapes are generally field models, and the majority of visualised components (projectiles, vehicles, people) are treated as entities steered by an event-driven visualisation framework. In real-world three-dimensional simulations, elements of the three visualisation types are combined in a *realism-calculation* balance to produce convincing representations of the world whilst retaining an efficient rendering cycle. Note that this balance does not necessarily mean that the accuracy of the underlying simulation model need be adjusted in all cases, but rather the realism quality of the visualisation layer.

Figure 16: A Visualisation of a Soviet T-28 Tank Created from Blue-Prints and Photographs.

Visual immersion is an important aspect of real-world simulation as these simulations are typically run to either involve a human interactively, or represent the actions of intelligent agents to a human audience - the value here is then psycho-visual. An excellent example of this type of application is in modern film production - the MASSIVE software has been used to simulate huge numbers of interacting intelligent agents as actors in the Lord of the Rings trilogy [53]. Visual immersion can be aided by creating accurate models of real-world entities to-scale. This is best done using three dimensional modeling software to build a mesh of the vehicle from real blue-prints. We can then use photos of the actual real-world object, taken from different angles, to accurately *texture* the model [54]. Our attempts at this process made use of the powerful, and freely available tools Blender3D and the GNU Image Manipulation Program (the GIMP) are illustrated in Figure 16. We have also made use of audio capture and manipulation software ffmpeg, and Audacity to record and reproduce the authentic vehicle sound effects of the simulated vehicles.

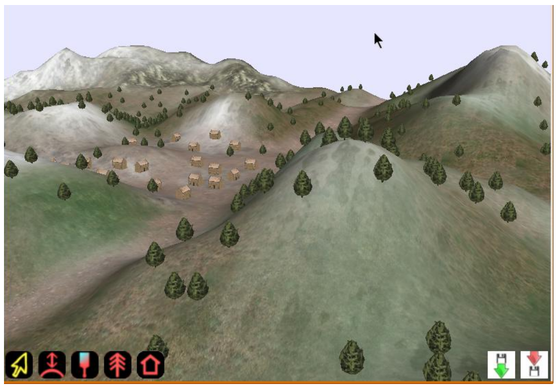
Figure 17: User-Assisted Dynamic Landscape Generation.

Most realistic terrain in 3D simulations is now generated by fractal modelers, which provide pseudo-realistic terrain but allow very little design control to scenario designers. The terrain illustrated in Figure 17, is however based on a novel method, in that it is visually created in 3D by hand, in real time, from within the simulation itself, with a variety of WYSIWYG ("What you see is what you get") tools that we have built into the simulator. This work is based on the ETM2 ("Editable Terrain Manager") for the OGRE graphics library [55], and gives us the added advantage that we can allow simulated elements in our event driven models to deform the landscape during simulation according to the physics model. We have, however, through OGRE, made use of algorithms such as ROAMing (Real-time Optimally Adapting Meshes) terrain [56] to optimally display the environment.

Combined visualisations can be quickly constructed by assembling an effective toolkit of different libraries and modules. There are now a huge variety of freely available open-source tools, libraries and high-level wrappers that offer very sophisticated visualisation technology. It is no longer the task of simulation builders to develop visualisation techniques for their simulations from the ground up, but rather to assemble a collection of these libraries and other resources that best suit the nature of their particular simulation project.

6 Simulation Software Tools

While simulation as an approach to computational science is a powerful method, it is only as good as the simulation models and the software that implements them. Many useful small scale simulations can be performed with existing packages such as problem solving environments like Matlab, Mathematica, Maple or even with simple calculations framework tools like spreadsheets.

However, for many application areas a custom simulation program is necessary to be able to handle large systems or novel algorithms. State of the art computational science typically needs state of the art simulation codes and the associated measurement and analysis codes.

As discussed elsewhere in this article, visualisation tools go hand in hand with simulation codes, both as a powerful tool for analysis and for gaining insights into the simulated systems, but also as a debugging and development tool. Often unusual effects in a simulation can be identified as artifacts of the simulation from an accompanying visual representation. In some cases the phenomena turn out to be genuine emergent features of the model and this can be very exciting to observe.

Simulation programs can be written in any appropriate programming language, but some languages and systems do lend themselves better than others to the requirements of:

- a clean and unambiguous syntax;
- some degree of portability in the sense of being able to attain repeatable and reproducible results from the same code on different hardware/operating systems platforms;
- support for and preferably pre-tested libraries of advanced data structures such as lists, sets, matrices and so forth.
- support for and existence of high level libraries for common simulations related programming tasks - typically numerical libraries for linear algebra, spectral transforms, random number generation, parameter fitting and so forth;
- a practical and preferably portable means of linking to a visualisation or rendering graphics library;
- and some means of optimising or at least addressing computational performance and memory requirements. In some cases a quality optimising compiler suffices, for other simulations a means of parallelising the algorithm and therefore support for parallel programming is required.

6.1 Packages and Custom Languages

Tools such as Matlab, Mathematica, Maple are often easier to use for simulations work by non-programmers although these tools typically support some sort of embedded high level programming language of their own. Our personal experience of these tools is that they are very good and useful up to a point. However, as general packages they may not be able to run a specific simulation model as fast as a custom code, and they may not necessarily use an optimal data representation for a particular problem. They are therefore often restricted to study of smaller model systems than a custom code might be able to tackle.

These tools are inevitably proprietary in nature and although portable across many hardware and operating systems platforms there are cost and licensing considerations in running a simulations based on them. Experience suggests they are excellent for rapid prototyping of simulations ideas and are often good learning tools for explaining small scale models.

These tools do represent a lot of systems development work and although there are some public domain alternatives such as GNU Octave, there is quite a wide variation in the different capabilities of these similar products and systems.

Like any proprietary or platform specific tool, there are also concerns about "lock in" and the difficulties in switching to a different tool set once a detailed simulation code has been developed. For a combination of this reason and those of performance, a custom simulation code developed in a general purpose programming language is often preferable.

6.2 General purpose Languages

Simulations can obviously be written in any programming language, but some languages have been traditionally more popular than others. As discussed one advantage of a custom simulation is that it exposes more opportunities for optimisation - either in terms of speed or memory and other resource utilisation. A great deal of scientific simulation code has been traditionally developed in Fortran and its various versions. Some noteworthy simulation code repositories for Fortran codes include the Computational Physics archives (CCP12, CCP5 - see `www.ccp5.ac.uk`).

Arguably modern Fortran versions such as Fortran 2005 do incorporate many modern language design features and software engineering support capabilities. Fortran's support for complex numbers is still an attractive feature for many simulations based on numerical algorithms. However it can also be argued that despite best efforts, Fortran syntax is struggling under the burden of modern language design ideas and it is not widely taught as a general purpose language.

Compiled languages based on the C syntax and semantics have become quite prevalent. C++ and Java are widely used for simulations development, and Java in particular with its strong portable graphics support is widely taught, widely used, and appears to have a relatively long term future. Newer variants such as the D language have absorbed many of the modern improvements in language design and might displace C, C++ and proprietary versions such as C# in the future.

Languages like Ada and Pascal have had important roles in custom simulation code development, but the sociological phenomena surrounding open development communities, widely available non- proprietary compilers and the widespread use of some languages for teaching purposes will likely continue to favour the C/Java syntactic family of languages.

Scripting and semi-interpreted languages such as Python [57] and Ruby [58] have made great inroads into the scientific community in recent years. It is certainly possible to use these languages for simulations work directly, although they will perhaps continue to find favour as integration languages for tying together fast simulation cores in C/C++ or Java. The Jython interface to Python is particularly helpful in this regard.

6.3 Libraries

A great many useful independent library projects arose during the Fortran era of scientific computation. Some well known examples include: Linpack [59]; Lapack; EISPACK [60]; the Harwell Subroutine Library; NAG Library and some well know library toolkits like the popular Numerical Recipes codes [61]. Generally those libraries had their origins with Fortran like languages and although successful ports or language interface layers have been developed for use with other languages such as C or Ada and Pascal, for the most part the ideas and structure embodied in many numerical libraries clearly show "Fortran thinking."

Similarly, modern graphical libraries typically trace their origins back to the need for Fortran bindings and some of course were written in Fortran. OpenGL and its predecessors still also show a certain amount of "Fortran thinking" too. Sun Microsystems' Java 3D is something of an exception and exhibits a rather different architectural structure more in keeping with modern language design, use of object orientation and high-level thinking. Perhaps surprisingly, this has not caught on in terms of user popularity as might have been expected and Java interfaces to OpenGL such as JOGL are sometimes preferred for some simulations purposes where performance optimisation needs to be fully exposed to the developer. We anticipate that this may be an important area for future development for the next generation of integrated and visualisation friendly simulation languages.

Typically a modern language will come with a standard development kit or library that will include all the numerical and visualisation components we have discussed above. Java is perhaps the best known example, and while the Java language itself is relatively compact, the associated libraries and development kit suite are quite complex and difficult for a programmer to retain in his mind at once.

Good quality libraries that make use of parallel programming ideas are still quite rare. The parallel and vector facilities in ScaLapack [62] are some of the few really good, usable parallel numerical libraries. This is another important area for future development as parallel computing algorithms return to the forefront for use in GPUs and multi-core processors.

6.4 Rendering Tools

Rendering tools fall into a number of different categories, all of which are typically necessary for simulations work in computational science. Graphical libraries that support 2D and 3D interactive-time rendering on screen are most useful for linking directly with custom simulation codes. As discussed there are many such libraries available, but the current most well known and useful are those based on the OpenGL interface [63]. The Java interface to this is known as JOGL [14] and is also easy to use and of high performance. Sun's newer library - Java3D [64] is powerful and also easily used from Java.

Post production rendering is often done using static configuration dumps from a simulation. Often tools such as physical rendering or ray-tracing are used for this. The most well known and public domain ray tracing tool is POVRay [65].

Other semi-graphical capabilities such as graphical interface widget libraries for buttons, checkboxes, sliders and other controls are also often necessary for interactive simulation codes. It is desirable that the chosen library be at least compatible or interoperable with the core graphics library. The Java Swing 2D [66] graphics and associated windowing tool kit is a useful such library. Other systems such as Tcl/Tk [67] are compatible with programming languages such as the C syntax language family and also scripting languages such as Python [57] and Ruby [58].

Finally it is important to have straightforward post production analysis tools for plotting data in **x-y** or **x-y-z** format. The most well known tool for this is Gnuplot [68] which has many fine features. Cartan [47] was developed as an interactive integrated plotting and parameter-fitting alternative.

Other data analysis can often be done with static images dumped to file, and useful toolkits for manipulation of 2D image data include the interactive tools xview [69] and GIMP [70] but also the PNM "anymap" image libraries [71].

7 Simulation Education

Computational scientists emerge from many different education routes. Recent interest in computational science as a discipline in its own right has raised the question as to whether there should be specific degree programmes or at least courses on computational science in general and simulation methods in particular [72]. It seems likely that for the foreseeable future many practitioners will undertake a typical high education degree through a traditional discipline be it computer science, applied mathematics, physics, chemistry, biology, engineering or some other applied area before possible specialisation in computational science at some sort of postgraduate level.

Arguably however there are important missed opportunities for teaching simulation techniques at undergraduate level and certainly for making them part of a final year project experience.

One noteworthy textbook on simulations lore and specific projects in physics applications is "An Introduction to Computer Simulation Methods" [10]. Another useful educational resource is the bi-monthly magazine "Computing in Science and Engineering", published by the IEEE Computer Society which contains reviews of various tools and methods and other practitioner and applications articles.

Many universities do offer specific courses on "computational methods" or "numerical methods" and provide project mechanisms for students to learn how to work in these areas. At present however it is probably true to say that most students will learn about simulations techniques in a rather *ad hoc* manner from working in or alongside research groups. Learning about graphics and rendering is likely to take place at the undergraduate level. Most Universities will offer some undergraduate course on computer graphics and some may also offer project experience using advanced rendering tools.

8 Summary and Conclusions

We have discussed a number of simulation paradigms and example applications. We have reviewed some of the complex systems issues facing simulation developers and users and have reviewed a number of the software tools and technologies that can be employed for a successful computational science experiment.

We believe that it is very helpful to have an advanced graphical rendering or visualisation capability that is closely coupled with a running simulation. This is an aid both to the simulation developer as a debug aid but also to the user for interpreting the simulation results and for steering the computations into relevant and interesting regions of the simulation model's parameter space.

As we have shown, the overall architecture can be driven from the simulation or agent perspective. A better understanding of a simulation's architecture can help to identify and avoid design-breaking features which would otherwise reduce the software's portability and re-usability. Visualisation is important in any simulation but plays different roles depending on the software goals.

It seems likely that a service-oriented approach to running simulations may become more prevalent but there will likely also remain a strong need for advanced custom crafted simulations as a tool for investigating state of the art complex systems.

The future holds some promise for the development of new tools and programming languages. One new system of note may be the Fortress combined language and environment [73]. Whatever tools are developed, there will be a continued need for close integration of simulation and rendering technologies that will further support portable, fast and ubiquitous simulations in computational science.

References

[1] Hawick, K., Wallace, D.: High performance computing for numerical applications. Technical Report EPCC-TR93-09, Edinburgh Parallel Computing Centre (1993) Keynote address, Proc. ACME Conference on Computational Mechanics, Swansea.

[2] Casti, J.L.: Would-Be Worlds: How Simulation is Changing the Frontiers of Science. Wiley (1996) ISBN-0471123080.

[3] Fox, G.C., Williams, R.D., Messina, P.C.: Parallel Computing Works! Morgan Kaufmann Publishers, Inc. (1994) ISBN 1-55860-253-4.

[4] Smarr, L., Catlett, C.E.: Metacomputing. Communications of the ACM **35** (1992) 44–52

[5] Hawick, K.: Computational science. Technical report, Massey University (2003) CSTN-000.

[6] James, H.A., Hawick, K.A.: Scientific data management in a grid environment. Journal of Grid Computing (2005) 1572–9814 ISSN: 1570-7873 (Paper) 1572-9814 (Online).

[7] Booth, S., Hawick, K.: Random Number generators for Super Computers. Technical Report ECSP-TN, Edinburgh Parallel Computing Centre, Edinburgh University, Mayfield Road, EH9 3JZ, UK (1991)

[8] Knuth, D.: The Art of Computer Programming: Seminumerical Algorithms. 3rd edn. Volume 2. Addison-Wesley (1997)

[9] C.F.Baillie, R.Gupta, K.A.Hawick, G.S.Pawley: Monte-Carlo Renormalisation Group Ising Calculations. Phys.Rev.B **45** (1992) 10438–10453

[10] Gould, H., Tobochnik, J., Christian, W.: An Introduction to Computer Simulation Methods. 3rd edn. Addison-Wesley (2006) ISBN: 0-8053-7758-1.

[11] Allen, M., Tildesley, D.: Computer simulation of liquids. Clarendon Press (1987)

[12] Newton, I.: Philosophiae Naturalis Principia Mathematica. apud Sa. Smith, London (1687)

[13] Hawick, K.A.: Domain Growth in Alloys. PhD thesis, Edinburgh University (1991)

[14] Bryson, T., Russell, K.: Java Binding for OpenGL (JOGL) (2007)

[15] T.A.Witten, L.M.Sander: Diffusion Limited Aggregation, a Kinetic critical Phenomenon. Phys.Rev.Lett. **47** (1981) 1400–1403

[16] Gardner, M.: Mathematical games: The fantastic combinations of john conway's new solitaire game "life". Scientific American **223** (1970) 120123

[17] Potts, R.B.: Some generalised order-disorder transformations. Proc. Roy. Soc (1951) 106–109 received July.

[18] Binney, J.J., Dowrick, N.J., Fisher, A.J., Newman, M.E.J.: The Theory of Critical Phenomena. Oxford University Press (1992)

[19] Lowry, M.C., Ashenden, P.J., A..Hawick, K.: A testbed system for time warp in java. Technical Report DHPC-087, University of Adelaide (2000)

[20] Barry, R.C., Chorley, R.J.: Atmosphere, weather and climate. 5 edn. Routledge (1989)

[21] Hawick, K.A., Bell, R.S., Dickinson, A., Surry, P.D., Wylie, B.J.N.: Parallelisation of the unified model data assimilation scheme. In: Proc. Workshop of Fifth ECMWF Workshop on Use of Parallel Processors in Meteorology, Reading, European Centre for Medium Range Weather Forecasting (ECMWF) (1992)

[22] W.E.Fitzgibbon, Wheeler, M.F., eds.: Computational methods in geosciences. SIAM (1992)

[23] Gonnella, G., Lamura, A., Sofonea, V.: Lattic boltzmann simulation of thermal nonideal fluids. Phys. Rev. E **76** (2007) 036703

[24] Allen, M.P.: Simulations Using Hard Particles. Royal Society of London Philosophical Transactions Series A **344** (1993) 323–337

[25] Sarman, S., Evans, D.J.: Heat flow and mass diffusion in binary lennard-jones mixtures. Phys.Rev.A **45** (1992) 2370–2379

[26] Donev, A., Torquato, S., Stillinger, F.H.: Neighbor list collision-driven molecular dynamics simulation for nonspherical hard particles: I. algorithmic details. J. Comput. Phys. **202** (2005) 737–764

[27] Playne, D.P.: Notes on particle simulation and visualisation. Hons. thesis, Computer Science, Massey University (2008)

[28] Ray, T.: An approach to the synthesis of life. Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity **xi** (1991) 371–408

[29] Adami, C.: On modeling life. In Brooks, R., Maes, P., eds.: Proc. Artificial Life IV, MIT Press (1994) 269–274

[30] Hawick, K.A., James, H.A., Scogings, C.J.: A zoology of emergent patterns in a predator-prey simulation model. In Nyongesa, H., ed.: Proceedings of the Sixth IASTED International Conference on Modelling, Simulation, and Optimization, Gabarone, Botswana (2006) 84–89

[31] Hawick, K.A., Scogings, C.J., James, H.A.: Defensive spiral emergence in a predator-prey model. Complexity International (2008) 1–10

[32] Hawick, K.A., James, H.A., Scogings, C.J.: Gridboxing for spatial simulation performance optimisation. In T.Znati, ed.: Proc. 39th Annual Simulation Symposium, Huntsville, Alabama, USA (2006) The Society for Modeling and Simulation International, Pub. IEEE Computer Society.

[33] James, H.A., Scogings, C.J., Hawick, K.A.: Parallel synchronization issues in simulating artifical life. In Gonzalez, T., ed.: Proc. 16th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS), Cambidge, MA, USA (2004) 815–820

[34] Hawick, K.A., James, H.A., Scogings, C.J.: Manual and semi-automated classification in a microscopic artificial life model. In: Proc. Int. Conf. on Computational Intelligence (CI'05), Calgary, Canada. (2005) 135–140

[35] Hawick, K., Scogings, C.J.: Resource scarcity effects on spatial species distribution in animat agent models. Tech Note CSTN-059, Computer Science, Massey University (2008)

[36] Hawick, K.A., James, H.A.: Simulating a computational grid with networked animat agents. In R.Buyya, T.Ma, eds.: Proc. Fourth Australasian Symposium on Grid Computing and e-Research (AusGrid 2006). CSTN-028, Hobart, Australia (2006) 63–70 ACSW Frontiers 2006, ISBN 1-920-68236-8, ISSN 1445-1336.

[37] Erdös, P., Rényi, A.: On random graphs. Publicationes Mathematicae **6** (1959) 290–297

[38] Hawick, K.A., James, H.A.: A taxonomic review of graph models and small-world networks. Technical Report Computational Science Technical Report CSTN-003, Massey University (2004)

[39] Newman, M.E.J.: The structure and function of complex networks. SIAM Review **45** (2003) 169

[40] Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393** (1998) 440–442

[41] Milgram, S.: The small world problem. Psychology Today **1967** (1967) 60–67

[42] Newman, M.E.J.: The structure of scientific collaboration networks. PNAS **98** (2001) 404–409

[43] Liljeros, F., Edling, C., Amaral, L., Stanley, H., Aberg, Y.: The web of human sexual contacts. Nature **411** (2001) 907–908

[44] Jeong, H., Tombor, B., Albert, R., Oltvai, Z., Barabsi, A.L.: The large-scale organization of metabolic networks. Nature **407** (2000) 651–654

[45] Fell, D.A., Wagner, A.: The small world of metabolism. Nature Biotechnology **18** (2000) 1121–1122

[46] Wagner, A., Fell, D.A.: The small world inside large metabolic networks. Proceedings of the Royal Society B **268** (2001) 1803–1810

[47] Hawick, K.: Cartan - a program for interactive model fitting, error analysis and visualisation. Technical Report CSTN-062, Computer Science, Massey University (2008)

[48] Watts, D.J.: Small worlds: the dynamics of networks between order and randomness. Princeton University Press (1999)

[49] Ubiety Lab: UbiGraph - a tool for visualising dynamic graphs. `http://www.ubietylab.net/ubigraph/` (2008) Last accessed August 2008.

[50] Hawick, K.: Interactive graph algorithm visualization and the graviz prototype. Technical Report CSTN-061, Computer Science, Massey University (2008)

[51] yWorks: yed - java graph editor. (2008)

[52] Smith, B.V.: Xfig Drawing Program for the X Windows System. (1996)

[53] Thalmann, D., Hery, C., Lippman, S., Ono, H., Regelous, S., Sutton, D.: Crowd and group animation. In: SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes, New York, NY, USA, ACM (2004) 34

[54] Ahearn, L.: 3D Game Art f/x & Design. Coriolis, Scottsdale, Arizona (2001) ISBN: 1-58880-100-4.

[55] Junker, G.: Pro OGRE 3D Programming. APress (2006) ISBN 1590597109.

[56] Duchaineau, M., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C., Mineev-Weinstein, M.B.: Roaming terrain: Real-time optimally adapting meshes. Technical Report UCRL-JC-127870, LLNL (1997)

[57] Guido van Rossum: Python essays. www.python.org/doc/essays (2008)

[58] Thomas, D., Fowler, C., Hunt, A.: Programming Ruby: The Pragmatic Programmer's Guide. 3rd edn. Pragmatic Programmers (2004) ISBN 978-0-9745140-5-5.

[59] Demmel, J., Dongarra, J., DuCroz, J., GreenBaum, A., Hammarling, S., Sorensen, D.: A project for developing a linear algebra library for high-performance computers. ANL-MCS-P37-1288 Preprint (1988)

[60] Bischof, C.H., Dongarra, J.J.: A linear algebra library for high-performance computers. In Carey, G.F., ed.: Parallel Supercomputing: Methods, Algorithms and Applications. Wiley (1989) 45–55

[61] Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C. 2nd edn. Cambridge University Press (1992) ISBN 0-521-43108-5.

[62] Choi, J., Dongarra, J.J., Pozo, R., Walker, D.W.: Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In: Proc. of the Fourth Symp. the Frontiers of Massively Parallel Computation, IEEE Computer Society Press (1992) 120–127

[63] Woo, M., Neider, J., Davis, T., Shreiner, D.: OpenGL Programming Guide: The Official Guide to Learning OpenGL. 3rd edition edn. Addison-Wesley (1999) ISBN:0201604582.

[64] Sowizral, H., Rushforth, K., Deering, M.: The Java 3D API Specification. 2nd edn. Sun Microsystems (2000) ISBN 978-0201710410.

[65] Persistence of Vision Pty. Ltd.: Persistence of vision (tm) raytracer. Williamstown, Victoria, Australia. (2004)

[66] Hoy, M., Wood, D., Loy, M., Elliot, J., Eckstein, R.: Java Swing. O'Reilly and Associates (2002) ISBN:0596004087.

[67] Welch, B., Jones, K., Hobbs, J.: Practical Programming in Tcl and Tk. 4th edition edn. Prentice Hall (2003) ISBN-13: 978-0130385604.

[68] Williams, T., Kelley, C., Lang, R., Kotz, D., Campbell, J., Elber, G., Woo, A.: Gnuplot command-driven interactive function plotting program. (2008)

[69] Heller, D.: XView programming manual: an open look toolkit for X11. O'Reilly and Associates (1990) ISBN:0-937175-38-2.

[70] Peck, A.: Beginning GIMP: From Novice to Professional. Apress (2006) ISBN 1-59059-587-4.

[71] Henderson, B., Poskanzer, J.: Netpbm - networked portable bit map image formats and tools. (2008)

[72] Blue Ribbon Panel on Simulation-Based Engineering Science: Revolutionizing engineering science through simulation. Technical report, National Science Foundation (2006)

[73] Allen, E., Chase, D., Flood, C., Luchangco, V., Maessen, J.W., Ryu, S., Jr., G.L.S.: Project fortress - a multicore language for multicore processors. Linux Magazine (2007) 38–43