Big Fast Crowds on PS3

Craig Reynolds
Sony Computer Entertainment, US R&D
craig_reynolds@playstation.sony.com
www.research.scea.com/pscrowd

Abstract

Crowds and other flock-like group motion are often modeled as *interacting particle systems*. These multi-agent simulations are computationally expensive because each agent must consider all of the others, if only to identify its neighbors. For large crowds, simple implementations are too slow since computation grows as the square of agent population. Faster approaches often rely on *spatial hashing* where a partitioning of space is used to accelerate crowd simulation. This same partitioning can form the basis of a scalable multi-processor approach to large, fast crowd simulations, as in [Quinn et al. 2003]. This paper describes an implementation of that approach for PLAYSTATION®3 which supports simulation and display of simple crowds of up to 15,000 individuals at 60 frames per second.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.6.8 [Simulation and Modeling]: Types of Simulation—Animation

Keywords: crowd simulation, multi-agent simulation, interacting particle systems, flocking, boids, behavioral animation, parallel processing, distributed processing, multi-processor

1 Introduction

This paper describes a technique for running large agent-based crowd simulations (interacting particle systems) using parallel processors to achieve high performance. It discusses an implementation for PS3, called PSCrowd, which distributes the simulation load across the Cell processor's multiple SPUs. Issues of agent behavior design and character animation will be touched upon but are not the central theme of this paper.

When building engaging virtual worlds, a key challenge is to keep them from looking like deserted "ghost towns." We want game worlds to be busy, complex and full of life, like a city bustling with pedestrians and vehicle traffic. Alternately a game might call for throngs of people at a fair or party, animals in a lush ecosystem or armies on a battlefield. We want our virtual worlds to be inhabited by thousands of autonomous characters (also known as *non-player characters*, NPCs). They must have plausible reactions to their environment and to other characters they encounter. When groups of characters meet, we expect them to interact, say by coordinating their motion, or by participating in other kinds of social interactions. Agent-based simulation is a common way to implement these autonomous characters to create crowds and other flock-like coordinated group motion. Agent-based models are ideal for capturing the nature of a crowd as a collection of individuals, each of which can have their own goals, knowledge and behaviors.

Because characters react to their neighbors, they must be able to identify neighbors by filtering nearby characters out of the whole population. The most direct way to do this is an $O(n^2)$ proximity screening: comparing each individual to all the others, collecting all those within a certain distance threshold. For crowd sizes up to a few hundred this approach is sufficient. For crowds of several thousand individuals, a more computationally efficient approach is required to allow simulation at interactive rates.

It has become common practice to accelerate the process of finding neighbors using some form of *spatial hashing* where individuals are pre-sorted by their approximate position. For example a regular grid can be overlaid on the world, individuals are assigned to the grid cell that contains their center point. To find all individuals within a given region, it is sufficient to consider those individuals assigned to cells which overlap the region of interest. The use of spatial hashing has become nearly ubiquitous for crowd/flock modeling as well as granular models of physical phenomena [Bell et al. 2005]. Other kinds of spatial hashing make use of quad/oct-trees [Shao and Terzopoulos 2005] and various useful partitioning schemes like *navigation meshes* (aka *navmesh*, see [O'Neill 2004] and [Miles 2006]).

On the hardware side, traditional single CPUs get incrementally faster. Multiple processors working in parallel provides a more direct path to higher performance. High end personal computers increasingly come equipped with dual processors. The Xbox console has three PowerPC processors. The PS3's Cell processor [Pham et al. 2005] contains one PowerPC processor and seven Synergistic Processor Units (SPUs). This trend is likely to continue with more and more independent processors being packaged together. To most effectively use these systems, software developers must recast their algorithms to use parallel computation. Ideally this is done in a way which is independent of the number of parallel processors, so the software can make use of whatever processors are available in a given architecture.

This paper describes a PS3-based multiprocessor algorithm for updating an agent-based crowd simulation. The algorithm uses a spatial partitioning both for spatial hashing and to divide the simulation update into disjoint *jobs* which can be evaluated in arbitrary order on any number of SPUs. A fine-grain partitioning suits SPU memory size and provides automatic load balancing.

2 Related Work

Applications of (non-interacting) particle systems were first described in the computer animation literature in [Reeves 1983]. However similar particle-based models had been used since the 1950s in the pioneering work on *computational fluid dynamics* by

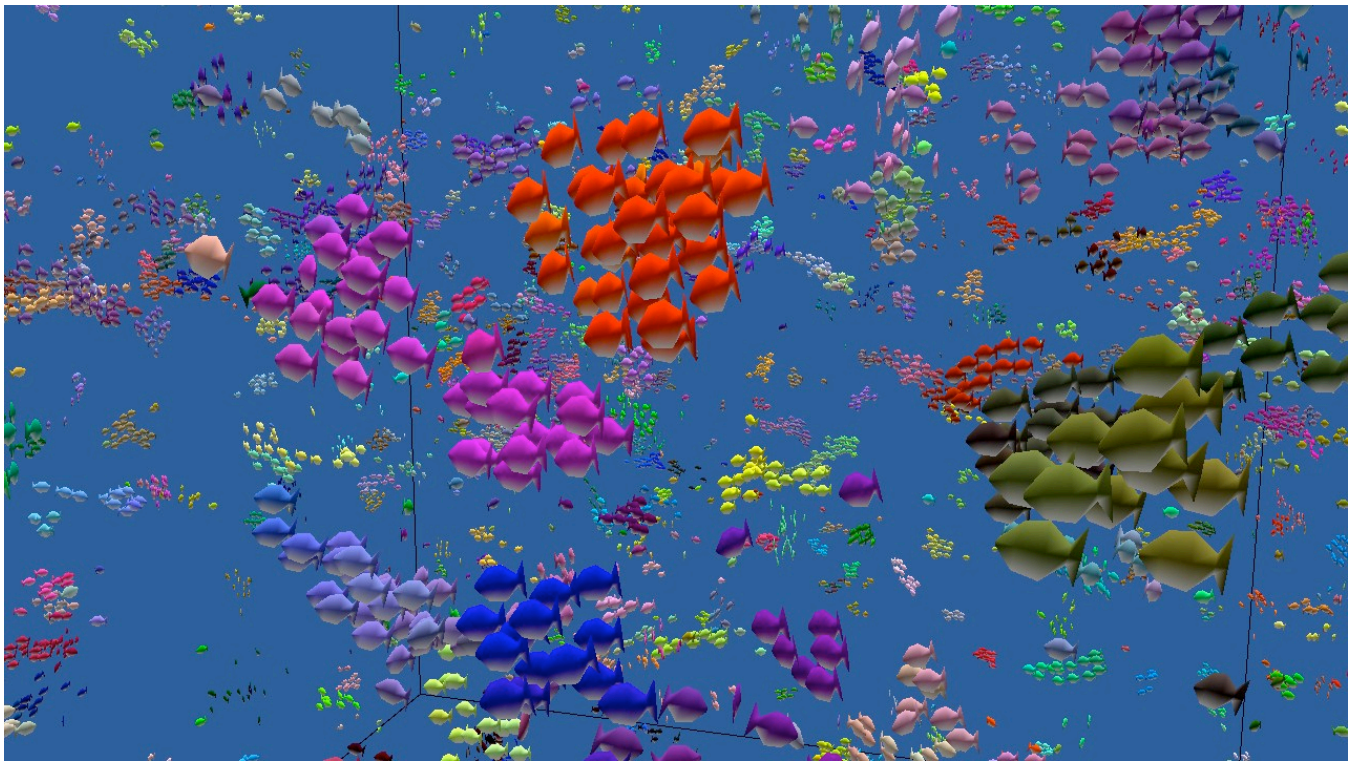

Figure 1: PSCrowd's Chameleon Fish demo, 10,000 schooling fish at 60 frames per second

Frank Harlow and his colleagues at Los Alamos National Labs. These included mesh/particle hybrid models (Particle-in-Cell (PIC) [Evans and Harlow 1957]), mesh-less free particle models (Particle-and-Force (PAF) [Harlow and Meixner 1961]) as well as several others. Today's widely used Lagrangian CFD methods, such as *smoothed particle hydrodynamics* [Gingold and Monaghan 1977] are closely related to interacting particle systems that underlie the crowd models discussed here. Use of parallel computation for CFD began in the late 1980s and [Sims 1990] described its application to particle system animation.

Agent-based simulation of flocking was described in [Reynolds 1987] which defined *boid* flocks in terms of interacting particle systems, noted the $O(n^2)$ bottleneck, and suggested spatial hashing as a solution. That approach was used in [Reynolds 1999] for an interactive flock simulation with 280 boids running at 60 fps (frames per second) on a PlayStation®2. The 1987 implementation was an off-line "batch" process, it took roughly one hour to simulate one second of flocking animation of 80 boids at 30 fps on a then state-of-the-art 1 MHz CPU.

While researching this paper I was pleasantly surprised to learn about a crowd model developed contemporaneously with boids. Frank Harlow, whose CFD work is cites above, realized that these Lagrangian models could be applied to *human collective dynamics*, see [Harlow and Sandoval 1986] and [Sandoval et al. 1988].

Helmut Lorek's work using using parallel computation to accelerate crowd/flock simulation began just a few years after the initial boids paper. [Lorek and White 1993] used a Meiko Transputer System with up to 50 processors to run a flocks of up to 100 boids at slow, but interactive rates. [Lorek and Sonnenschein 1995] used a CM-5 to run Huth and Wissel's 1992 model of fish schools.

Using techniques from the field of GPGPU (general purpose computation on graphics processing unit) two groups built systems where crowd or flock simulations were computed using a combination of CPU and GPU. The FastCrowd system [Courty and Musse 2005] ran a crowd of 5000 individuals at about 100 fps, and a crowd of 10,000 at 35 fps (without visualization, 50 fps and 20 fps with individuals drawn as 2D disks). The GPU also computed the flow of smoke for fire evacuation scenarios. The GEBs system described in [Erra et al. 2004] could simulate a flock of 1600 boids at 60 fps, with 8000 boids the system ran at about 20 fps. These rates include rendering a 3d scene with animated bird models. GEBs also used a novel optimization, a *scattering matrix* to detect when the flock departs from mainly parallel flight.

A very high performance engine for interacting particle systems called Outburst (originally Kinema/Sim) was commercialized by Animation Science (originally ArSciMed). They also produced a 2.5D version for crowd simulation called Rampage (originally Kinema/Way). Technical aspects of these systems are described in [Bouvier et al. 1997]. While typically used on single CPU systems, section 2.2.4 of that paper describes an implementation using PVM (parallel virtual machines). It used one dimensional partitioning for spatial hashing and dynamic load balancing. (A very similar partitioning was used in [Zhou and Zhou 2004] for a cluster of up to 16 networked Linux PCs, running a flock of up to 512 boids.)

Distributed multiprocessors are used in [Quinn et al. 2003] to run large evacuation scenarios involving 10,000 pedestrians at 45 fps on 10 processors of the SWARM cluster connected by a gigabit Ethernet switch. As described below, this work is very similar to PSCrowd. Differences between the two are motivated by the underlying hardware, which differ most significantly in memory size of the parallel processors and data transmission rates between them. (Note: the paper says 10,000 at 45 fps. One of the authors, Ron Metoyer, told me in email it was 80 fps with no graphics.)

The crowd model in [Treuille et al. 2006] (to appear at SIGGRAPH 2006) is a unique hybrid of a fluid flow model and a crowd model. A continuum field is created each frame to globally characterize the crowd and environment, then individual agents navigate according to this field. On a fast PC, it can run a simulation with 10,000 agents at 5 fps without graphics. The simulation rate is 2 fps with graphics, but that provides a thread displaying interpolated frames showing humanoid characters at 12 fps.

In [Tecchia et al. 2001] members of the simulated crowd do react to each other albeit with a fairly simple behavioral model (individuals simply rotate until they found a collision free path). Running on a single processor PC this system could achieve 37 fps with 5000 individuals and 21 fps with 10,000 individuals. This performance includes the load of rendering an urban setting and animated humanoid representations of each individual.

The autonomous pedestrian model in [Shao and Terzopoulos 2005] exhibits both high performance and sophisticated goal-driven behavior models of people at a train station. Without graphics they can simulate 1400 pedestrians at 30 fps on a modern PC. With humanoid character animation and rendering of the complex environment results in rates of 3.8 fps for 500 individuals. Similarly, [Pelechano et al. 2005] presents a detailed model of high density crowds in emergency building evacuation scenarios, with cognitive modeling of agent knowledge, communication and psychology. Without graphics it simulates 1800 agents at 25 fps.

Some other works on closely related topics: [Steed and Abou-Haidar 2003] assumes a crowd simulation is running on several networked servers and that moving an individual from one server to another is an expensive operation. It investigates how to find the best static partitioning for the environment based on traffic density statistics. While Lagrangian CFD is the focus of [Frank et al. 2001] its analysis of static and dynamic spatial partitionings (which it calls SDD and DDD) is directly applicable to crowd simulation. It compares the two techniques on two different applications for several multiprocessor systems. [Merchant et al. 1998] compares two static and three dynamic partitionings for individual based models. [Plimpton 1995] compares several approaches to parallel computation for short range molecular dynamics, which is essentially equivalent to crowd simulation.

3 Interacting Particle Systems and Parallelism

In general, any two particles can react to each other in an interacting particle system. More commonly, particles are restricted to *local* interactions so the "behavioral kernel" has *finite support*. This leads to computational efficiencies, and can serve as a model of the localized perception that would be provided to a realistic agent by its own senses. In the local case we can assume that two particles will have no effect on each other if they are more than a certain distance apart.

Limiting particles to local interactions in combination with spatial hashing allows significant speed-up of the simulation. Finding all particles in a given neighborhood no longer requires considering all other individuals. We need consider only those in the hashing partitions which overlap the neighborhood. Depending on the analysis and assumption used, these accelerated interacting particle systems can be considered either $O(n)$ [Quinn et al. 2003] or "just barely" quadratic (see Figure 10 of [Shao and Terzopoulos 2005] where the factor on the n^2 term is 0.000229).

To apply multiple processors to a simulation update, the workload must be divided into jobs that can be executed independently and in parallel. Ideally the jobs would also be order-independent to avoid scheduling restrictions. In some implementations particles are simply divided into static arbitrary groups which are updated together (one of the options examined in [Plimpton 1995]). In the approach taken here, particles are dynamically grouped based on their position. All particles in a certain region of space are updated together. This grouping is based on the granularity of the spatial hashing scheme already in place to accelerate neighbor-finding. (Conversely, if the hashing scheme is not a pure spatial partitioning the approach in this paper may not be applicable.)

When a simulation is updated on multiple processors, information must flow from one processor to another. Depending on the multiprocessor architecture, data may be transmitted over a network connection, IO channel or DMA bus. To facilitate this communication, data structures used in the simulation must be *compact* and *self-contained.* Specifically the use of pointers should be avoided in mobile data structures. This issue is significant because many spatial hashing implementations make extensive use of pointers.

4 PS3 Implementation

PSCrowd uses the same subdivision for three purposes: hashing 3D space, managing memory layout, and assigning processors to jobs. The spatial hashing used in the current implementation is a static, regular 3d lattice of box-shaped voxels, as in Figure 2. Each voxel is represented by a C++ class called *Bucket*. The *Lattice* class contains a three dimensional array of Buckets. Members of the crowd (the particles) are represented by a base class called *Individual*. All Individuals within a Bucket are stored in a compact array-like data structure, allowing easy transfer by DMA to or from an SPU's local memory. In one *job* of the multiprocessor update algorithm, all Individuals in one Bucket are updated by one SPU (with read-only reference to surrounding Buckets for perception of the local neighborhood). Because the Buckets are spatially disjoint, they can be updated in parallel and in any order.

This implementation was developed to make effective use of the hardware architecture of the PS3's Cell processor [IBM et al. 2005] and its RSX GPU. The PS3 Cell has a 3.2 GHz clock speed. It contains one Power Processor Unit (PPU—a standard PowerPC CPU) and seven Synergistic Processor Units (SPUs). These processors live inside *elements* on the Element Interconnect Bus, which also talks to the Memory Interface Controller and I/O controllers. This path is very fast, DMA between elements and the 256 Mbyte XDR system memory achieves a peak rate of 25.6 GBytes/sec. On PS3 one SPU is normally reserved, so the current version of PSCrowd uses up to 6 SPUs working in parallel to update the simulation. The update process can be shared among

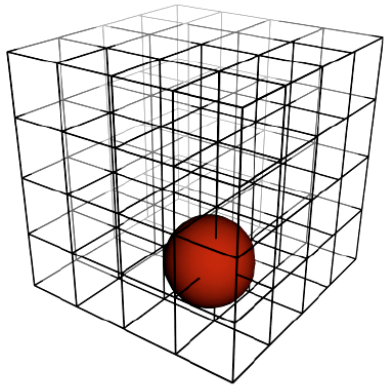

Figure 2: spherical neighborhood within a Lattice of Buckets

any number of available SPUs. The Cell's SPUs provide very fast execution, but their local store holds only 256 Kbyte. Typically processing on an SPU involves shuttling data to and from main memory using DMA.

The next three sections of this paper will describe the data structures (C++ classes) used in this implementation, provide a summary of the multiprocessor simulation update cycle, and discuss some design considerations.

4.1 Implementation Objects

Some implementation details about the C++ classes used in PSCrowd are listed below. Note that Individual is used as a base class and that Bucket, Lattice (etc.) are template container classes parameterized by an application-specific class derived from Individual. Because instances of these classes move between PPU and SPUs, they are compiled for both processor architectures. Interprocessor communication in PSCrowd consists of transferring these C++ objects between main XDR memory and SPU memory using DMA. Accordingly these classes are defined to be aligned on the 128 byte boundaries which are most efficiently handled by Cell's DMA.

Individual: an instance of this class represents one member of a crowd. It is intended as a base class whose functionality is inherited by an application-specific class of individual. (For example, the demos supplied with PSCrowd define a class called *Fish* that is derived from Individual.) Individual-based classes provide their own per-agent, per-frame *update* function. (For the PSCrowd demos they also provide various per-crowd utilities as static class functions.) An Individual instance contains position and orientation information, as well as speed and body radius information. They also each have a unique ID number. A class derived from Individual, like Fish, will include additional state information related to its specific behavior and animation.

Bucket: a template container class for a collection of Individual-based instances. It corresponds geometrically to an axis-aligned box in 3D space. All Individuals whose center-point falls within a given box are stored inside the corresponding Bucket's instance. The instance consists of some header information and an array of Individual-based instances. It is important to note that this is the only copy of an Individual's definition, it is not a pointer to, or a temporary copy of, static data stored elsewhere. In the current implementation Buckets are all the same fixed size, and so have a fixed maximum capacity. (Hence an undesirable failure mode of PSCrowd: "Bucket overflow." Simulation cannot proceed if a Bucket's storage capacity is exceeded.) As Individuals move they will cross the boundary from one Bucket to another. To ensure that each Individual remains assigned to the correct Bucket, a *rebucket* operation is applied once per frame on the PPU. Each

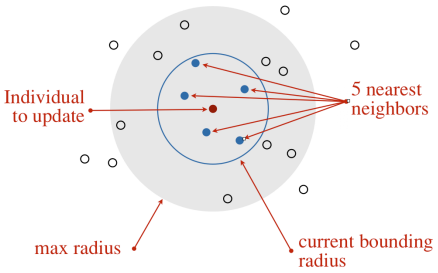

Figure 3: finding the NearestN neighbors

Individual is rehashed: a new Bucket index is computed from its position. If the new Bucket is different from the old, the Individual is deleted from the old Bucket and added to the new. Both of these operations use a constant time $O(1)$ algorithm. New Individuals are added to the end of the active array, based on a stored size. To remove an Individual (given its index) it is overwritten with a copy of the last Individual in the active array, then the Bucket size is reduced by one. As a result, data representing an Individual moves within Buckets, from Bucket to Bucket and from processor to processor. A pointer to an Individual is valid at most for one simulation step. So keeping track of some particular Individual in the crowd (say to follow it with the camera) becomes problematic. See Section 4.3 for the approach taken here.

Lattice: this class serves as the central control for the whole simulation. It contains all of the Buckets, which contain all of the Individuals. Correspondingly Lattice is a template of a class based on Individual. The Buckets in a Lattice are identical in size and are arranged in a 3D array. They are allocated in main memory, though Buckets move via DMA to and from SPUs for update.

NearestN: this object holds the state of a search for the N nearest neighbors of an Individuals position. This kind of search is also called "K nearest neighbors." It is defined by: a position, a maximum radius and N. All Individuals in nearby Buckets (those which intersect the given spherical neighborhood) are passed into the NearestN object for consideration. It builds an ordered collection of the N nearest neighbors within given sphere, as shown in Figure 3.

CondensedIndividual, CondensedBucket: when a Bucket's Individuals are updated by an SPU, read-only reference is made to certain properties (primarily position and heading) of Individuals in neighboring Buckets. (These boundary Individuals are called *ghosts* in [Quinn et al. 2003].) To save memory space on the SPU, condensed copies of Buckets containing condensed copies of Individuals are cached at the beginning of each frame.

BucketUpdateParameters (BUP): this object mediates communication between the PPU and each SPU job. It is shared (via DMA polling) between the two processors. Synchronization is provided by two flags in the BUP: *ready* and *done*. The PPU waits (spins) until some BUP is done, fills in the BUP, then sets *done* to false and *ready* to true. The SPU repeatedly reads its BUP via DMA and begins work when the BUP is ready. After the Bucket update is complete the SPU sets *ready* to false and *done* to true then DMAs the BUP back to main memory. At the end of the per-frame simulation update, the PPU waits until all BUPs are done.

Because of the large number of Individuals in PSCrowd simulations, the cost of drawing all of them is significant, even when very simple geometrical models are used to represent their bodies. PSCrowd uses the (OpenGL|ES based) PSGL graphics library [Arnaud 2006] in conjunction with the Cg programming language [Nvidia 2006] to achieve high performance graphics with a tight coupling to the crowd simulation process.

If a crowd consists of many nearly identical characters, the use of graphical *instancing* allows a substantial savings in graphics data volume and so much improved GPU cache usage. Instancing is supported in PSGL and Cg by allowing the user to specify a *parameter element function* to access graphical data using division or modulus by instance size to modify the element index. Together these can be used to provide a Cg *vertex program* with vertices from the shared geometry and per-instance parameters for a given Individual. For example, the demos described in Section 5 use a single fish-shaped body for all Individuals as in Figure 1. This body geometry, as indexed triangle vertices, is sent to the RSX GPU just once, in a data structure known as a VBO (vertex

buffer object). Another VBO contains per-instance parameters, one set for each Individual in the simulation. In the case of the fish demos described below, the per-instance data consists of a transformation matrix, a color and a swim-cycle phase. The vertex program "customizes" the shared geometry by the per-instance parameters. The per-instance parameter VBO is filled during PSCrowd's simulation update and is double buffered to allow drawing of one frame to overlap with simulation of the next.

4.2 Simulation Update Cycle

In PSCrowd the PPU controls the main per-frame update cycle. It executes some once-per-frame operations itself and synchronizes communication with the SPUs for simulation update:

- For each Bucket: make a CondensedBucket copy
- For each Bucket: assign the next free SPU to update it.
- Wait for: all SPUs done, draw done, and v-sync.
- Draw instances stored in VBO during update. Swap VBOs.
- Rebucket: reassign Individuals who cross Bucket boundary

Regarding drawing and waiting: issuing the main draw command using Cg and PSGL API, initiates the process (transfer VBO with per-instance data to the RSX, kick off rendering) and then returns. The double buffered VBOs are flipped, and the next simulation step begins to fill the "other" VBO. Opening (mapping) a VBO for write will wait until any previous draw is completed. This rarely comes up because of double buffering. The V-sync feature prevents drawing from getting more than one frame ahead, and so can cause a wait.

During each simulation update, each Buckets is updated by one SPU. In the abstract these updates are independent and could happen in parallel. In reality, PSCrowd simulations involve thousands of Buckets to be updated by six SPUs. As a result the update is quasi-parallel. Typically at any one time, six SPUs are working on six Buckets. When an SPU finishes its update, it is assigned another Bucket, chosen sequentially from the array of Buckets. In the current implementation the PPU handles this serialization. After all Buckets have been assigned to SPUs, the PPU waits for all SPUs to finish.

From the perspective of an SPU its job consists of updating just a single isolated Bucket:

- DMA job data to SPU from XDR main memory:
  - poll BUP until ready, then:
  - the Bucket to be updated
  - its 26 neighboring CondensedBuckets (9 in 2D)
- Update all Individuals in center Bucket, see Figure 4:
  - refer to neighboring CondensedBuckets
  - store per-instance data in local "VBO chunk" buffer
- DMA per-instance data into VBO mapped into RSX memory
- DMA updated Bucket back to XDR main memory

The current version of PSCrowd supports only limited character animation. It does not support the kind of articulated figure animation that would typically be used to represent walking human characters, like those used in [Shao and Terzopoulos 2005], [Tecchia et al. 2001] or [Treuille 2006]. In the demos described in Section 5, the characters are fish. Beyond simple rigid motion expressed by a transformation matrix, the only other animation they exhibit is a swimming motion for the fish's tail. That tail swishing motion is provided procedurally in the Cg vertex program used to transform the vertices of the fish body model by each instance transform.

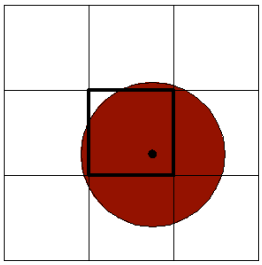

Figure 4: in 2D, a circular neighborhood on a 3x3 grid of Buckets, centered on an Individual inside the central Bucket. An SPU updates the central Bucket, making read-only reference to the surrounding CondensedBuckets.

4.3 Design considerations

Between PSCrowd's Bucket update jobs, no simulation state remains resident on an SPU. In other crowd systems, designed for other platforms, the number of spatial partitions is sometimes set equal to the number of processors (called *servers* in [Steed and Abou-Haidar 2003] and *worker processes* in [Quinn et al. 2003]). This allows simulation state to reside on the parallel processors, distributing memory load and reducing communication costs. That approach was not feasible in PSCrowd because of the small size of an SPU's local store. Instead PSCrowd stores simulation state in main memory, moving it temporarily to an SPU for processing, then right back to main memory. This approach better suits the Cell architecture because main memory is large, SPU memory is small, and DMA is very fast.

Load balancing is a central theme of many multiprocessor-based systems for interacting particle systems. For example, [Steed and Abou-Haidar 2003] describes a static scheme based on a priori map knowledge. Many authors have proposed dynamic schemes to balance loads [Bouvier et al. 1997; Merchant et al. 1998; Frank et al. 2001; Zhou and Zhou 2004]. This focus on load balancing may be due to the relatively coarse partitioning of space that these systems use, to allow pairing partitions with processors, to allow resident data, to avoid high communication costs. When processors are permanently assigned to specific partitions of space, their load will necessarily fluctuate as particles clump, forming non-homogenous, time-varying densities across partitions. In contrast, PSCrowd does no explicit load balancing. The load on each SPU is kept roughly equal because of the finer partitioning of space used by PSCrowd. This property of having many Buckets, updated by a handful of processors, was motivated by an SPU's small local store. It has the beneficial side effect of dicing the simulation volume into many samples, some crowded and some empty, allowing the load to naturally balance across the SPUs.

As mentioned in Section 4.1, the data representing an Individual is mobile: it can move within its Bucket and move from Bucket to Bucket. The Buckets themselves move between main memory and SPU local stores. As a result, a pointer to an individual is not a reliable way to identify or find it. Originally, to allow tracking an Individual (for a "chase camera," annotation or other applica-

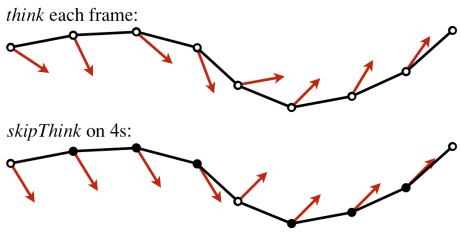

Figure 5: *skipThink* applies same steering for multiple frames
top: agent path, behavioral steering computed each frame
bottom: steering computed every Nth frame and repeated

tions) PSCrowd ran a linear search on the PPU to find the individual with a given ID number. As crowd sizes continues to grow, this implementation became prohibitive. Later a special mechanism was added to the SPU-based update procedure. As an SPU is updating the Individuals in a Bucket, it checks each against a list of "special" ID numbers. When one is found, the SPU DMAs a copy of the Individual back to a known static location in main memory, where it can be referenced for graphics or game logic.

Frame rates between 30 and 60 fps produce smooth, vivid animation. Behavioral updates can happen much less frequently if they are decoupled from the animation rate [Reynolds 2000]. As long as gross position (particle physics) and character animation proceed at the higher rate, slower behavioral updates produce few noticeable artifacts. In PSCrowd demos, the physics, animation and graphics run at 60 fps while behavioral updates are made every eighth or tenth frame (so at 7.5 fps to 6 fps). A similar approach is taken in [Treuille et al. 2006] where the crowd simulation for 10,000 agents runs at 2 fps while animation proceeds at 12 fps (24 fps for smaller agent populations). In PSCrowd this mechanism is called skipThink. On any given frame, 1/8 of Individuals "think" (find neighbors and compute a behavioral steering force) while 7/8 of Individuals skip thinking and apply the steering force computed on the last think frame, as shown in Figure 5.

5 Notes on demos and behaviors

PSCrowd is distributed as a runtime library together with sample application code to demonstrate its use. These demos currently include: Chameleon Fish, Fish Species and a simple Crowd demo (see video recordings: **http://www.research.scea.com/pscrowd/**). All three demos run at 60 fps. In the first two, 10,000 fish-like Individuals are free to move in 3D within a cube shaped Lattice measuring 308 "units" in each dimension (producing a density of 0.00034 fish per cubic unit) divided into 2744 (14x14x14) Buckets. Each Bucket can contain up to 160 Fish. The fish bodies have a bounding sphere radius of 2.4 units. In the third demo 15,000 Individuals are constrained to move on a 2D ground plane, using a Lattice divided into 2500 (50x1x50) Buckets. The first two demos use a skipThink count of 8 and the 2D crowd uses skipThink of 10. In all three cases, Individuals are controlled by variations on the boids model [Reynolds 1987]. A significant difference is that as in [Erra et al. 2004], PSCrowd considers only the 5 nearest neighbors while steering each boid. In the 1987 version, *all* boids within a given neighborhood were considered during steering computations.

In addition to basic boids behavior (consisting of *separation*, *alignment* and *cohesion*) each fish uses other steering behaviors. *Obstacle avoidance* prevents collisions with the Lattice's bounding box and other obstacles present in the environment. *Leader wander* adds variety and allows flocks to break apart. *Anti-vertical* makes sure fish do not swim too steeply up or down, see Figure 6. Finally *anti-crowding* is a self-preservation measure for the simulation: fish in crowded Buckets try to spread out to avoid exceeding the fixed upper bound on Bucket population.

The Chameleon Fish also exhibit *flock coloring*, a chameleon-like visual behavior where each fish's body mimics the color of neighbors just ahead of them. As a result the members of local subflock clusters tend to have similar colors. In the Fish Species demo, the fish have two sets of flocking parameters, one for members of their own species and one for all others. They tend to congregate and swim with members of their own species.

Running a traditional boid flock simulation inside a smooth container tends to eventually produce one large flock. Initially individual boids meet and form small "flocklets." Those meet and merge into larger flocks, and eventually all individuals are swept together into one large group. This and other undesirable consequences of the tradition boids model have been carefully analyzed in [Bajec et al. 2005]. Bajec notes that because a boid's perceptual neighborhood is forward-looking, individuals on the leading edge of a flock have little or no input. As a result they fly a boring straight path, which is then imitated by boids behind it. Using Bajec's definition we will consider these boids that perceive no neighbors to be temporary emergent *leaders* and the others to be *followers*.

The *leader wander* behavior used in PSCrowd's demo substitutes random *wander* steering for leaders, including solitary fish, who would otherwise steer straight ahead. As a result, leaders take occasional random turns that followers may imitate. This can be thought of as a simplistic model of the leader's perceptual and cognitive processes. Followers do not wander. They steer as accurately as they can to coordinate with their neighbors. Similarly *flock coloring* "wanders" the color of leaders, which is imitated by followers, so diverging groups take on slightly different "team colors." Because different leaders may wander in different directions, this behavior occasionally causes flocks to split. The PSCrowd demos are tuned so schools of fish grow and shrink easily, resulting in a large number of small schools whose size distribution stays consistent over many hours of simulation time.

6 Results

As described above, crowd system performance is a multidimensional quantity. There are costs for large simulations and there are separate costs for animation of thousands of animated characters. Systems differ by the complexity of agent behavior, the graphical sophistication of the individual bodies and by the complexity of the environment. The underlying hardware differs from system to

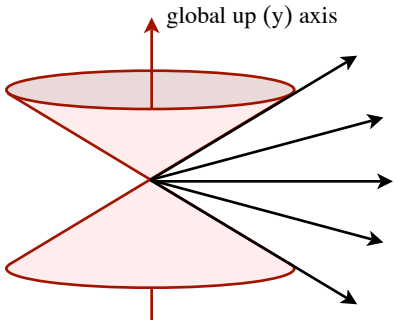

Figure 6: anti-vertical Fish behavior,
heading is constrained to be outside of cone.

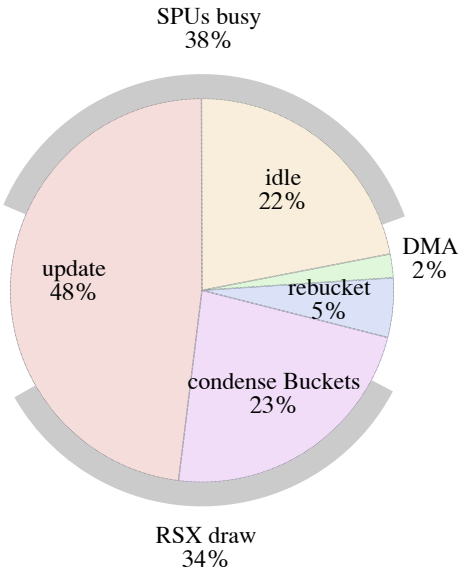

Figure 7: for Chameleon Fish demo, 10,000 fish at 60 fps
center chart: distribution of PPU time spent per frame
top: portion of frame SPUs are busy
bottom: portion of frame RSX GPU is busy

system. All these factors make it hard to do apples-to-apples performance comparisons of different systems. For the same reason, any performance statistics quoted for a crowd systems requires a lot of explanation of just what is being measured.

PSCrowd can produce crowds of up to 15,000 individuals at 60 fps including both simulation and graphics. These results were measured while running on a PS3 development system (DEH-R1015 with SDK 0.8.4) generating video output at HD 720p. The the 2D Crowd demo mentioned in Section 5 supports 15,000 individuals at 60 fps. The 3D demos (Chameleon and Species) run slower, they can handle 10,000 of the same characters at 60 fps. (Measured another way: with a population of 10,000 for all three cases, the average load, expressed in compute time as a percentage of 1/60 second: 2d crowd: 70%, chameleon: 80% and species 90%) Beyond skipThink differences, another factor is that in the 2D case, the Bucket neighborhood is 3x3 while in 3D it is 3x3x3. Three times as many CondensedBuckets must be processed during each 3D Bucket update. Also, when agents with separation behavior are restricted to a 2D surface they tend to be less densely packed per unit volume, hence per Bucket.

The animated "fish bodies" used in these demos have limited geometric detail. They are somewhat more complex than representing individuals as points or disks, but much less detailed than an animated human character in a typical video game. Specifically the fish bodies contain 20 indexed vertices and 36 triangles. The bodies have a swimming motion in the tail provided by displacement in the Cg vertex program.

The breakdown of processor utilization for the Chameleon Fish demo is shown in Figure 7. The PPU spends about half of each frame on simulation update, about a quarter of the frame creating CondensedBuckets, about 5% on rebucket, and just 2% is spent on DMA. This leaves about 22% of the frame time idle to provide *headroom* for occasional slow frames. The PPU "update" cost includes assigning each idle SPU a Bucket to update, waiting for one of them to complete, and so on for all Buckets. On average the SPUs are busy for only 38% of each frame. The RSX GPU is busy for only 34% of the frame drawing the simple fish bodies.

7 Future Work

The previous section indicated that the PPU spends half its time coordinating with the SPUs, the SPUs and RSX are idle about 2/3 of each frame. So one might reasonably ask: why is PSCrowd *so slow?* The current limitations are that: adding Buckets will increase the PPU update cost, and adding density (Individuals per Bucket) will increase the memory load on the SPU, which already hovers near 99%. Exploring how to change PSCrowd to take advantage of this additional performance will be a focus of future work. Tapping the unused RSX power is easy, it simply requires more complicated geometric modeling and advanced rendering techniques as shown in Figure 7 from Phil Harrison's GDC 2006 Keynote. Those fish have bodies composed of up to 400 triangles (plus two lower levels of detail for each of three species). The scene uses realistic underwater lighting and haze effects. High dynamic range lighting refracts through an animated water surface. Gabor Nagy wrote the software for graphics, rendering, procedural water and art path. Plus he hooked PSCrowd into his framework while I was off on vacation. Care Michaud-Wideman created all of the art for this project. This PSCrowd simulation of 5000 fish plus all the underwater rendering effects runs at 30 fps.

The approaches taken in PSCrowd and in [Quinn et al. 2003] are very similar, except for design decisions based on differences in the underlying hardware. PSCrowd deals with very fast communication and very small local memories. Quinn et al. deals with slower communication and larger memories. Future versions of PSCrowd may take a hybrid approach, supporting multiple Cell processors on a network, creating fast local "islands" of SPUs connected by slower external communication channels.

Other topics for future work include PSCrowd's large footprint in main memory. Because all Buckets have a fixed size, only a part of which is normally in use, the Lattice that contains them is very sparse. It is roughly 50 times bigger than it would need to be if each bucket was dynamically reallocated to fit its current contents of Individuals. This might be done by recasting Buckets as a partitioning of a large array of Individuals. Removing the fixed Bucket size would also prevent "Bucket overflow" a failure mode which is avoided now only by a priori tuning.

Rather than try to fit 27 (3x3x3) Buckets into an SPU's local store, it might work better to *stream* the neighboring Buckets (condensed, or maybe not) through the SPUs memory during a Bucket update job. This would reduce the Bucket storage requirement from 27 to 2, but would require additional storage for a NearestN object for each Individual in the center Bucket.

Collision avoidance between Individuals in the current demos is weak, especially for head-on collisions between groups. When behavioral avoidance occasionally fails, There is no physical or kinematic non-penetration constraint to keep fish from passing through each other. Finally PSCrowd should be generalized to handle spatial partitioning schemes other than the finite, regular, box-shaped partitioning it uses now. For example it should support navmeshes and BSP or KD trees. Another interesting topic is abstract spatial hashing schemes which are not based on compact partitions [Teschner et al. 2003].

8 Conclusion

This paper has described the PSCrowd library for running high performance crowd simulation and animation on PS3. While sharing many aspects of other high performance crowd systems,

PSCrowd is able to capitalize on unique aspects of its platform to produce very large, fast crowds.

Fast hardware increasingly means parallel hardware. High performance software is increasingly hardware specific. Programming around hardware leads us in new directions. An interesting aspect of this work is that SPU memory limitations lead to use of many spatial partitions updated by a handful of processors. A side effect of this is automatic load balancing "for free."

This work was sponsored by my employer, Sony Computer Entertainment. It has been supported by many colleagues in Japan, Europe and California. I particularly wish to thank my US R&D coworkers: Gabor Nagy, Care Michaud-Wideman, Roy Hashimoto, Axel Mamode, Steven Osman, Stewart Sargison, Tanya Scovill, Trevor Smigiel, Chengdong Li, Greg Corson and Nicholas Szeto. Special thanks to to my manager, Dominic Mallinson, Director of Technology and head of US R&D.

I wish to thank Lance Williams for suggesting I connect the history of this work to the seminal CDF work at Los Alamos. Special thanks to Dr. Frank Harlow, who kindly and patiently explained some of that early work to me via email. Finally, thanks to my supportive family for giving me time to focus on projects like this.

References

ARNAUD, R. 2006. PSGL (PlayStation Graphics Library). GDC 2006 presentation. http://www.khronos.org/developers/content/GDC_2006/GLESTutorial07-PlayStation_GL.pdf

BAJEC, I. L., ZIMIC, N. AND MRAZ, M. 2005. Simulating flocks on the wing: the fuzzy approach. *Journal of Theoretical Biology* 233(2), 199–220. DOI= http://dx.doi.org/10.1016/j.jtbi.2004.10.003 http://lrss.fri.uni-lj.si/people/ilbajec/papers/ilb_jtb05.pdf

BELL, N., YU, Y., AND MUCHA, P. J. 2005. Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM Siggraph/Eurographics Symposium on Computer Animation* (Los Angeles, California, July 29 – 31, 2005). SCA '05. ACM Press, New York, NY, 77–86. DOI= http://doi.acm.org/10.1145/1073368.1073379

BOUVIER, E., COHEN, E., AND NAJMAN, L. 1997. From Crowd Simulation to Airbag Deployment: Particle Systems, a New Paradigm of Simulation. *Journal of Electronic Imaging* 6(1), 94.107 http://www.esiee.fr/~najmanl/papers/particlesystem.pdf

COURTY, N. AND MUSSE, S. R. 2005. Simulation of large crowds in emergency situations including gaseous phenomena. In *Proceedings of IEEE Computer Graphics International 2005*, 206-212. DOI= 10.1109/CGI.2005.1500417 (see http://home.tele2.fr/ncourty/fastCrowd.htm)

ERRA, U., DE CHIARA, R., SCARANO, V., TATAFIORE, M. 2004. Massive Simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. *Proceedings of Vision, Modeling and Visualization 2004 (VMV)*. http://wonderland.dia.unisa.it/projects/gebs/

EVANS, M.W. AND HARLOW, F.H. 1957. *The particle-in-cell method for hydrodynamic calculations.* Los Alamos Scientific Laboratory report LA-2139

FRANK, T., BERNERT, K. AND PACHLER, K. 2001. Dynamic Load Balancing for Lagrangian Particle Tracking Algorithms on MIMD Cluster Computers. In *PARCO'2001 – International Conference on Parallel Computing 2001*. Naples, Italy, September 4–7, 2001, pp. 1–9. http://www.imech.tu-chemnitz.de/mpf/publication/frank/parco_2001.ps.Z

GINGOLD, R. A., MONAGHAN, J. J. 1977. Smoothed particle hydrodynamics – Theory and application to non-spherical stars.

Figure 7: 5000 high detail fish at 30 fps with LOD, underwater haze, animated water surface, HDR illumination

Royal Astronomical Society, Monthly Notices. Vol 181, 375-389. http://adsabs.harvard.edu/abs/1977MNRAS.181..375G

HARLOW, F. H. AND MEIXNER, B. D. 1961. *The particle-and-force computing method for fluid dynamics.* Los Alamos Scientific Laboratory report LA-2567-MS.

HARLOW, F. H., AND SANDOVAL, D. L. 1986. Human Collective Dynamics: The Mathematical Modeling of Mobs. In *Mathematical Modeling of Biological Ensembles,* Los Alamos National Laboratory report LA-10765-MS.

LOREK, H. AND WHITE, M. 1993. Parallel Bird Flocking Simulation. http://citeseer.ist.psu.edu/lorek93parallel.html

LOREK, H. AND SONNENSCHEIN, M. 1995. Using parallel computers to simulate individual-oriented models in ecology: A case study. In *Proceedings of the 1995 European Simulation Multiconference* (ESM), 526–531, June 1995. http://citeseer.ist.psu.edu/lorek95using.html

IBM, SONY AND TOSHIBA. 2005. *Cell Broadband Engine Architecture.* Version 1.0, August 8, 2005. http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA277638 7257060006E61BA/$file/CBEA_01_pub.pdf

MILES, D. 2006. Crowds In A Polygon Soup: Next-Gen Path Planning. GDC 2006 presentation. http://www.babelflux.com/gdc2006_miles_david_pathplanning.ppt

MERCHANT, F., BIC, L. AND DILLENCOURT, M. B. 1998. Load Balancing in Individual-Based Spatial Applications. In proceedings of the *International Conference on Parallel Architectures and Compilation Techniques* (IEEE PACT 1998). 350–357. http://citeseer.ist.psu.edu/merchant98load.html

NVIDIA CORPORATION . 2006. *Cg Toolkit User's Manual.* Release 1.4.1 http://developer.nvidia.com/object/cg_toolkit.html

O'NEILL, J. C. 2004. Efficient Navigation Mesh Implementation. *Journal of Game Development* 1(1) 71–90.

PHAM, D., et al. (20 authors). 2005. The Design and Implementation of a First-Generation CELL Processor. In *Solid-State Circuits Conference, 2005. ISSCC 2005 IEEE International.* http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/7FB9EC5D5BBF51ED87256FC000742186/$file/ISSCC-10.2-Cell_Design.PDF

PELECHANO, N., O'BRIEN, K., SILVERMAN, B., BADLER, N. 2005. Crowd Simulation Incorporating Agent Psychological Models, Roles and Communication. *Workshop on Crowd Simulation* (V-CROWDS 2005), 24–25. http://www.seas.upenn.edu/~npelecha/Pelechano_V_CROWDS05.pdf

PLIMPTON, S. 1995. Fast Parallel Algorithms for Short Range Molecular Dynamics. *Journal of Computational Physics*, 117, 1, pages 1–19. http://citeseer.ist.psu.edu/plimpton95fast.html

QUINN, M. J., METOYER, R. A., AND HUNTER-ZAWORSKI, K., 2003 Parallel Implementation of the Social Forces Model. In *Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics* (August 2003), pp. 63–74. http://eecs.oregonstate.edu/gait/pubs/QuinnFinal.pdf

REEVES, W. T. 1983. Particle Systems—a Technique for Modeling a Class of Fuzzy Objects. ACM Trans. Graph. 2, 2 (Apr. 1983) 91–108. DOI= http://doi.acm.org/10.1145/357318.357320

REYNOLDS, C. W. 1987. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and interactive Techniques* M. C. Stone, Ed. SIGGRAPH '87. ACM Press, New York, NY, 25-34. DOI= http://doi.acm.org/10.1145/37401.37406

REYNOLDS, C. W. 2000. Interaction with Groups of Autonomous Characters, in proceedings of the *Game Developers Conference 2000*, CMP Game Media Group, San Francisco, California, 449–460. http://www.red3d.com/cwr/papers/2000/pip.html

SANDOVAL, D. L., HARLOW, F. H., AND GENIN, K. E. 1988. Human Collective Dynamics: Two Groups in Adversarial Encounter. *Los Alamos National Laboratory report LA-11247-MS.*

SHAO, W. AND TERZOPOULOS, D. 2005. Autonomous pedestrians. In *Proceedings of the 2005 ACM Siggraph/Eurographics Symposium on Computer Animation* (Los Angeles, California, July 29–31, 2005). SCA '05. ACM Press, New York, NY, 19–28. DOI= http://doi.acm.org/10.1145/1073368.1073371

SIMS, K. 1990. Particle animation and rendering using data parallel computation. In *Proceedings of the 17th Annual Conference on Computer Graphics and interactive Techniques* (Dallas, TX, USA). SIGGRAPH '90. ACM Press, New York, NY, 405-413. DOI= http://doi.acm.org/10.1145/97879.97923

STEED, A. AND ABOU-HAIDAR, R. 2003. Partitioning crowded virtual environments. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (Osaka, Japan, October 01 – 03, 2003). VRST '03. ACM Press, New York, NY, 7–14. DOI= http://doi.acm.org/10.1145/1008653.1008658

TECCHIA, F., LOSCOS, C., CONROY, R., AND CHRYSANTHOU, Y. 2001. Agent behaviour simulator (ABS): A platform for urban behaviour development. In *Games Technology 2001* (GTEC 2001) Hong Kong, China. http://www.cs.ucl.ac.uk/staff/Y.Chrysanthou/crowds/papers/TecchiaGTEC2001.pdf see also: http://www.cs.ucl.ac.uk/staff/Y.Chrysanthou/crowds/sketch/

TESCHNER, M., HEIDELBERGER, B., MUELLER, M., POMERANETS, D. GROSS, M.. 2003. Optimized spatial hashing for collision detection of deformable objects. *Proceedings of Vision, Modeling and Visualization* (VMV 2003), pages 47–54. http://citeseer.ist.psu.edu/teschner03optimized.html

TREUILLE, A., COOPER, S. AND POPOVIĆ, Z. To appear 2006. Continuum Crowds (in preparation for proceedings of SIGGRAPH 2006.) *ACM Trans. Graph.* 25(3) http://grail.cs.washington.edu/projects/crowd-flows/

ZHOU, B. AND ZHOU, S. 2004. Parallel Simulation of Group Behaviors. In *Proceedings of the 2004 Winter Simulation Conference*. Volume 1, pages 364–370 http://www.informs-cs.org/wsc04papers/043.pdf