Multi Agent Navigation on the GPU

Avi Bleiweiss

NVIDIA Corporation

ableiweiss@nvidia.com

Abstract

We present a unique and elegant graphics hardware realization of multi agent simulation. Specifically, we adapted Velocity Obstacles that suits well parallel computation on single instruction, multiple thread, SIMT, type architecture. We explore hash based nearest neighbors search to considerably optimize the algorithm when mapped on to the GPU. Moreover, to alleviate inefficiencies of agent level concurrency, primarily exposed in small agent count (≤32) scenarios, we exploit nested data parallel in unrolling the inner velocity iteration, demonstrating an appreciable performance increase. Simulation of ten thousand agents created with our system runs on current hardware at a real time rate of eighteen frames per second. Our software implementation builds on NVIDIA's CUDA.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search— Graph and Tree Search Strategies; I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence—Intelligent Agents

Keywords: graphics hardware**,** agent, search, navigation, velocity obstacles, multi threading, binary tree, hash, multiprocessor, kernel

1 Introduction

Multi agent systems have been recently gaining increased attention by game AI developers, mainly in the area of motion planning for non player characters. Simply stated, the principal challenging problem here is the safe navigation of an agent to its goal location, avoiding collision with both other moving agents and with static or potentially dynamic obstacles. Then, to subscribe efficiently on the GPU, a plausible planner solution must exercise decomposable movement [Li and Gupta 2007], and more importantly, scale well to environments with hundreds and thousands of individual agents. Of course, it must perform at interactive rates for excessively dense settings.

Velocity Obstacles (VO) [Fiorini and Shiller 1998] is a generally applicable, well defined and simple technique that has been widely used for safely navigating agents among moving obstacles [Shiller et al. 2001; Kluge and Prassler 2007; Fulgenzi et al. 2007]. VO represents a set of agent velocities that would result in a collision with an obstacle that moves at a certain velocity, at some future time. Whereas the complement set of avoidance velocities is intersected with a set of admissible velocities to produce dynamically, feasible maneuvers for the agent. A new VO space is computed in regular, discrete time intervals and the path from a start to a goal position is derived by searching potential velocities to either minimize distance or travel duration. VO solves well the pattern of passively moving obstacles that progress with little or no awareness of their surroundings.

In a multi agent formation agents must however be aware of each other and constantly yield by adapting their trajectory to avoid collision. Agent awareness of others is confined to the knowledge of their current state represented by position and velocity. An agent becomes cognizant of other agents (or obstacles for that matter) when they fall inside its field of view. VO was identified marginal when deployed in multi agent setups and prone to produce undesirable oscillatory motion [Feurtey 2000]. Devised VO extensions [Abe and Yoshiki 2001; Kluge and Prassler 2007] provided less than an optimal solution to address VO instability until the introduction of Reciprocal Velocity Obstacles (RVO) [Van Den Berg et al. 2008-ICRA]. RVO embraces the reactive behavior of other agents [Krishna and Hexmoor 2004] assuming all agents make similar collision free reasoning.

Main Contributions. This paper presents the work and the challenges we overcame in porting the RVO technology [Van Den Berg et al. 2008-I3D] on to the GPU. Our model extends RVO to remarkably improve simulation scalability by replacing a naïve nearest neighbors search with a hash based, and by refitting the algorithm to expose deeper parallelism. We demonstrate credible speedup compared to a sixteen threaded, CPU implementation, for crowds of up to several tens of thousands agents. Finally, we provide comprehensive analytical profile for GPU resource usage, memory access patterns and system level performance.

2 Related Work

The continuum crowds [Treuille et al. 2006] model unifies global path planning and local collision avoidance by using a set of dynamic potential and velocity fields that guide individual motion simultaneously. Fickett and Zarko [2007] compute the 2D potential field on the GPU using a tile based approach. The potential field is constructed using an eikonal solver [Jeong and Whitaker 2007] that runs on graphics hardware seven times faster compared to the accomplished marching algorithm, referenced in the original work. They further devise exploiting temporal coherence to drive computation down by only updating tiles of interest. Nonetheless, the footprint of the potential field, especially with proposed extensions to address higher space dimensionality, introduces a relatively large, sparse data structure in video memory and exhibits suboptimal locality of reference for fairly random spatial queries.

Human motion capture data offers an alternative for effectively creating new animations [Treuille et al. 2007]. Essentially, interpolating and concatenating dynamic clips into a realistic script. In his thesis, Karthikeyan [2008] offers an animation framework that uses the GPU to render crowds of virtual humans, in real time. The technique utilizes motion graphs [Kovar et al. 2002] for splicing an existing database of short animation sequences, and produces a continuous and a much longer clip for rendering. This implementation uses CUDA [Nvidia 2007] and

runs the GPU as a general compute device. It demonstrates a significant animation speedup compared to an equivalent CPU implementation. However, this approach is less scalable and suffers from an increased space complexity of the motion graphs that must be stored in video memory for its entirety.

Our work was directly inspired and builds on top of RVO. RVO produces smooth, visually compelling motion in setups formed by crowds with hundreds and thousands of agents. But foremost, each agent navigates individually without explicit communication amongst agents, and all pursue identical, free collision reasoning. RVO is thereby an embarrassingly parallel workload proving performance scale on the CPU, nearly linear with the number of agents. Extremely attractive is its intuitive formulation of integrating global path planning and local navigation, leading to a compact and economic memory model. At the same time, RVO is notably computationally intensive, but this is well aligned with a teraflop power, capable GPU.

3 Multi Agent Navigation

The overarching force driving navigation planning is that agents have a destination or a goal. We consider goal selection an external parameter set by the game developer. Barring environmental conditions, we assume agents move to their goal in the fastest speed possible. Most importantly, the presence of other moving agents affect speed and in the extreme case a pair of agents cannot intersect each other. In general, agents choose the minimal distance path to their destination, but even when they move unobstructed they abide by a preferred global path they constantly consult. Presently, we compute the global path on the CPU in a onetime, preprocessing step. This involves the reduced visibility graph [LaValle 2006] method to obtain a shortest path roadmap that prohibits agents from contacting static obstacles. Then, our navigation model, composed of agents, obstacles and a roadmap, is discretized in time and simulation advances all agents per timestep, concurrently. Noteworthy in our framework is a dynamic obstacle entity that is treated as a specially tagged agent, passively moving in a constant, unaltered velocity.

3.1 Visibility

We can only offer here a short summary of roadmap related tasks, namely visibility and shortest path. We define a visible point with respect to a vantage point, an observer, if the line that connects the points does not intersect any of the static obstacles present in the scene. To simplify computation and without loss of generality, input polygonal obstacles are each further reduced to a set of boundary line segments. Then, the visibility processing step generates two sets of edges, E0 that connects pairs of visible roadmap nodes and E1 linking the goal position of each agent to unblocked roadmap nodes. For resolving roadmap connectivity we favored an efficient implementation of the sweep plane algorithm [Shamos and Hoey 1976]. This amounts to finding the set of intersections of a moving vertical line, originating in either a roadmap node or in an agent goal, with line segment obstacles. Let n be the number of obstacle segments, the running time complexity of the algorithm is $O(n\log n)$ and space limit is of $O(n)$. We conclude the simulation setup step by invoking the Dijkstra [Russel and Norvig 1995] search algorithm to determine the shortest path from an agent goal position to any of the roadmap nodes. Both the visibility and shortest path computations described ideally suit a SIMT engine. In fact, recent work for running roadmap search algorithms on the GPU [Bleiweiss 2008] confirms noticeable performance gains and we expect appreciable scale in porting visibility processing on to the hardware. The fairly large visibility data structures produced here require though a specialized video memory layout for compelling multi thread access, further discussed in section 4.

3.2 Mathematical Model

We will now briefly address the mathematical model for collision avoidance dynamics. The VO governing equation follows:

$$VO_B^A(\boldsymbol{v}_B) = \{\, \boldsymbol{v}_A \mid \lambda(\boldsymbol{p}_A, \boldsymbol{v}_A - \boldsymbol{v}_B) \cap B \oplus -A \neq \emptyset \,\}.$$

Let A be an agent with reference point $\boldsymbol{p}_A$ and B a disc shaped obstacle centered in $\boldsymbol{p}_B$. Let $\oplus$ denote a Minkowski sum and λ a ray defined by an origin and a direction. $VO_B^A(\boldsymbol{v}_B)$ (Figure 1(a)) is the VO of obstacle B to agent A defining a set of velocities $\boldsymbol{v}_A$ for A that result in collision with obstacle B moving in velocity $\boldsymbol{v}_B$, at some later point in time. The Minkowski difference of a line segment shaped obstacle B $\{\, \boldsymbol{p0}, \boldsymbol{p1} \,\}$, used in our model, and a disc A is an extended parallelogram shown in Figure 1(b).

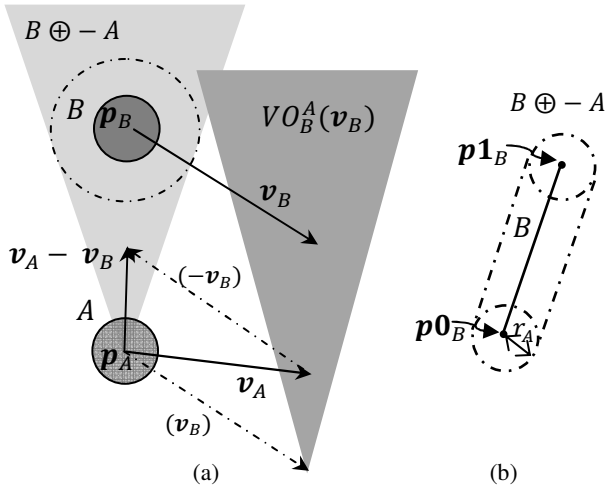

Figure 1**:** Velocity Obstacle $VO_B^A(\boldsymbol{v}_B)$ of disc shaped, obstacle B to disc shaped, agent A (a); Minkowski difference of line shaped, obstacle B $\{\, \boldsymbol{p0}, \boldsymbol{p1} \,\}$, and disc shaped, agent A (b).

RVO extends VO to overcome potential oscillatory movement [Van Den Berg et al. 2008-ICRA]. Rather than choosing for each agent a velocity that is outside the other agent's VO, the new velocity per timestep is an average of the agent current velocity and a velocity that lies outside the VO of the other agent. The formalization of RVO is outlined in the following equation:

$$RVO_B^A(\boldsymbol{v}_B, \boldsymbol{v}_A) = \{\, \boldsymbol{v'}_A \mid 2\,\boldsymbol{v'}_A - \boldsymbol{v}_A \in VO_B^A(\boldsymbol{v}_B) \,\}.$$

$RVO_B^A(\boldsymbol{v}_B, \boldsymbol{v}_A)$ is the RVO of agent B to agent A and is defined as a set consisting of all velocities $\boldsymbol{v}_A$ for agent A that will result in a collision with agent B at future point in time, assuming that agent B chooses a velocity $\boldsymbol{v}_B$ in its own specified RVO. RVO geometrical interpretation is the translation of the collision cone $VO_B^A(\boldsymbol{v}_B)$ to the apex $\frac{\boldsymbol{v}_A + \boldsymbol{v}_B}{2}$, as illustrated in Figure 2. RVO is further generalized and adds an inter agent, collision reasoning load factor α_B^A, in the range of 0 to 1, that expresses the agent rule of reciprocity: $\alpha_B^A = 1 - \alpha_A^B$. The form of geometrical translation now places the VO cone apex in $(1 - \alpha_B^A)\boldsymbol{v}_A - \alpha_B^A \boldsymbol{v}_B$. Lastly, in a multi agent setup the combined RVO for agent A_i is the union of

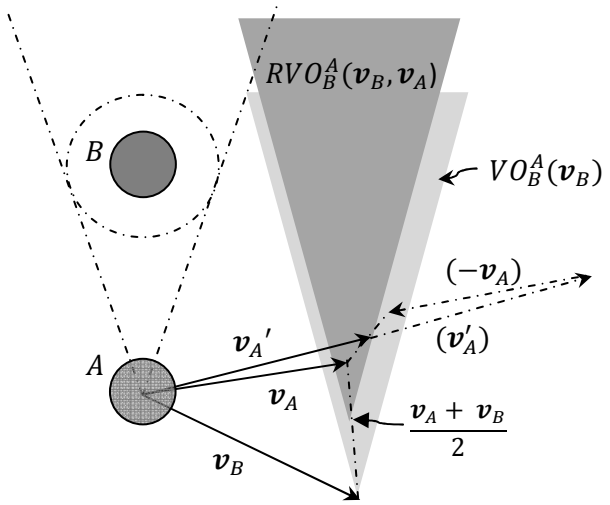

Figure 2: Reciprocal Velocity Obstacle $RVO_B^A(\boldsymbol{v}_B, \boldsymbol{v}_A)$ of agent B to agent A, translating the collision cone $VO_B^A(\boldsymbol{v}_B)$ to the apex $\frac{\boldsymbol{v}_A + \boldsymbol{v}_B}{2}$.

all RVOs created by other agents and the VOs generated by all the obstacles in the scene:

$$RVO^i = \cup_{i,j} RVO_j^i(\boldsymbol{v}_j, \boldsymbol{v}_i, \alpha_j^i) \cup \cup_{O \in O} VO_O^i(\boldsymbol{v}_O).$$

The problem of multi agent simulation can be now formulated and reduced to searching an optimal agent velocity in a set of permissible velocities outside the claimed union of RVOs, for each time interval.

3.3 Simulation

In this section we formalize the steps for simulating our model. A simulation session runs until all agents reach their goal or until a maximum limit of running time exceeds a preset system cap. Our simulator advances through each timestep performing parallel computation on all agents in a three stage pipeline, hash (optional, discussed later), simulate and update, as depicted in Figure 3:

```
1:  VO = velocity obstacle
2:  RVO = reciprocal velocity obstacle
3:  do
4:     hash
5:        construct hash table
6:     simulate
7:        compute preferred velocity
8:        compute proximity scope
9:        foreach velocity sample do   ← nested parallel
10:          foreach neighbor do
11:             if OBSTACLE then VO
12:             elseif AGENT then RVO
13:          resolve new velocity
14:    update
15:       update position, velocity
16:       resolve at-goal
17: while not all-at-goal              ← flat parallel
```

Figure 3: Pseudo code for simulator advancement through each timestep with hash (optional), simulate and update phases.

In the first step of the simulate phase we compute a preferred velocity vector. Its magnitude is bound to a preferred speed value set externally, and its direction faces from the current agent position to either a roadmap node or to the agent goal. The closest roadmap node is any of the visible nodes from another roadmap node or from the agent goal, or a node of the shortest path tree, computed as part of the visibility preprocess described in a previous section. Ideally, we would want the new velocity selected for the agent to be as close to the preferred velocity.

In the next step we compute the agent proximity scope. We assert that simulating the model described in the previous section for every possible pair of agents and any combination of agent and obstacle is unnecessary and would require excessive amount of computation. The basis for our claim is that an agent has less knowledge of a far away agent intention and it is thus less likely for them to affect the motion of each other. We thereby adaptively compute a limited scope of neighboring agents and obstacles, confined to a certain distance from the agent of concern. The agent proximity scope is further bound by a programmer set maximum number of the closest, and most likely visible neighbors.

We then select the best new velocity for the agent from the set of velocities outside the combined RVO. In our model agents are subject to dynamic constraints that include maximum speed v_i^{max} and maximum acceleration a_i^{max}. The new velocity search is additionally qualified then to be in the following admissible set:

$$AV^i(\boldsymbol{v}_i) = \{\, \boldsymbol{v'}_i \mid ||\boldsymbol{v'}_i|| < v_i^{max} \wedge ||\boldsymbol{v'}_i - \boldsymbol{v}_i|| < a_i^{max} \Delta t \,\},$$

where Δt is the timestep. We arrive at the best new velocity by evaluating number of random samples evenly distributed over the set of allowable velocities. Where the number of velocity samples is a global simulation parameter set externally. The quality of a candidate velocity is affected by its deviation from the preferred velocity and by the inverse of the time-to-collision with the agent neighbor. This is further expressed in the following cost function:

$$cost_i(\boldsymbol{v'}_i) = ||\boldsymbol{v}_i^{pref} - \boldsymbol{v'}_i|| + w_i \frac{1}{time-to-collision(\boldsymbol{v}_i)},$$

where $\boldsymbol{v}_i^{pref}$ is the preferred velocity and w_i is a weighting factor that controls the agent aggression. Note the time-to-collision for no foreseen collision is infinity. For each candidate velocity we look for the minimum time-to-collision in the combined RVO, generated for the agent neighboring scope, and select the new velocity to be the candidate velocity with the minimal cost:

$$\boldsymbol{v}_i^{new} = min\ cost_i(\boldsymbol{v'}_i)\ where\ \boldsymbol{v'}_i \in AV^i.$$

The algorithm described addresses crowded scenes with their combined RVO space likely filling up the entire set of admissible velocities. While picking up velocities inside the union of RVOs is an incidental liability, it was empirically proven to be resolved in progressive updates. Also, in theory, the navigation model ensures that no pair of agents will intersect. However, in practice we enforce a pair-wise minimum distance that we flag once being violated. Then, in selecting the new velocity we rather compute the time-from-collision and make agents too close part away.

The update phase is computationally light weight and is invoked synchronously once the new velocity has been resolved for all the engaging agents. We first scale the velocity of the agent to obey maximum acceleration and follow with an update to the agent position. We then resolve the at-goal agent state by checking the distance between the updated position and the goal to be within an

externally set, goal radius parameter. Finally, once all agents have reached their goal, simulation terminates.

4 Implementation

Multi agent simulation on the GPU presents several implementation challenges. Most importantly are (a) hiding global memory latency [Buck et al. 2003], (b) mitigating thread divergence, (c) minimizing hash table constructing cost and (d) efficient thread safe, random number generation (RNG). Our simulator operates on all agents simultaneously and is governed by a pair of simulate and update CUDA kernels. Optionally, we fork off nested, velocity sample iteration by launching a set of independent thread grids, one per agent (Figure 3).

The communication paths of our CUDA simulator are straight forward. In each simulation step the GPU provides back to the CPU main simulator thread a list denoting at-goal status, per agent. The list cumulative state thus makes up for our session termination criteria. In addition, the GPU emits two arrays, one of positions destined for visualizing the agent computed waypoints; and one of velocities for interacting with the physics simulator of a game engine. Further, deeper discussions of AI, physics simulation integration is outside the scope of this paper. We now look more closely at some GPU unique, simulator design considerations to confront our challenges.

4.1 Data Layout

We have all our simulator data structures reside in global memory. Static per timestep and any modifiable data structures are kept in non cached, read-only or read-write global memory locations. Although it would be intuitive to store visibility and shortest path data as an array-of-structures (AoS) this has serious memory access implications with severely reduced bandwidth due to thread unaligned, data layout. Instead, we store roadmap related data in a more efficient collection of structure-of-arrays (SoA). By grouping thread related data in contiguous arrays we improve substantially the possibility of coalesced memory transactions across a half-warp.

We maintain two identical visibility data structures, one for roadmap nodes and one for agent goals. Per node or goal vertex, visibility data is split into a pair of vectors, one listing unobstructed, angular vertex view of static obstacles in the form of $\{ angle, segment\_id \}$ and the other is a collection of distances from the vertex to a roadmap node $\{ distance, node\_id \}$. The elements of the vectors are of type $\{ float, int \}$, each taking 8 bytes in global memory. The vectors are further aggregated into a collection of vectors as depicted in Figure 4. Vectors inside a collection are of arbitrary length and are indexed or iterated using a pair of linear offset and count parameters. Then, in video memory the visibility hierarchy is represented as a pair of collection of vectors and a top level array listing $\{ offset, count \}$ for specific vector dereferencing, for each the roadmap nodes and the agent goals.

A third array enlists agent goal shortest path trees, each formatted as a vector of distances from a roadmap node to an agent goal$\{ distance, node_id \}$. The shortest path data structure follows identical visibility data access pattern illustrated in Figure 4.

Read-only resources throughout the simulation session are stored as a set of linear device memory regions bound to texture references. They include global simulation controls (Figure 5) and static obstacles, each represented as a line segment occupying a pair of four component texels, one for each vertex. With the benefit of being cached, texture potentially displays higher bandwidth for localized access.

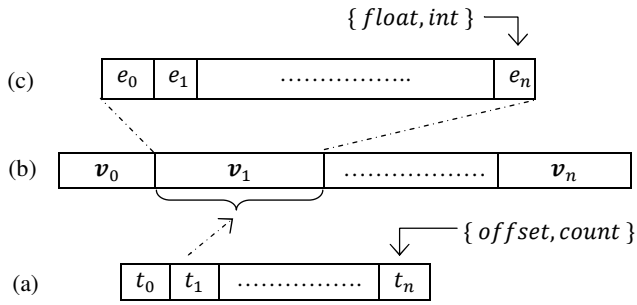

Figure 4: Visibility and shortest path data hierarchy. The index array (a) is thread aligned and uses a pair of $\{ offset, count \}$ to locate a vector of a roadmap node or a goal vertex in the array of vectors (b), vector elements are of type $\{ float, int \}$ (c).

```
struct CUSimulation {
   float timestep;
   struct  proximity; { maxneighbors,  distance  }
   float velocitysamples;
};
```

Figure 5: CUDA global simulation controls data structure, stored in a single texel texture.

The CUDA agent data structure shown in Figure 6 is the focal resource for communicating simulation results across kernels. It groups float4 vector members and internal data structures, each composed of 4 byte, scalar variables of no more than 16 bytes, padded as necessary. This yields a top level, 16 byte aligned data structure in memory that is highly effective for performing vector member accesses. In our simulator model vector data types were made extensible and more forward looking. They are each of three or four, 32 bit float or integer components, for computation and storage, respectively.

```
struct CUAgent {
   float4 start, goal;
   struct  shape; { radius, goalradius }
   float4 position, velocity;
   float4 prefvelocity, potential, candidate;
   struct constraint; { perfspeed, maxspeed, maxacceleration, safety }
   struct cost; { timetocollision, minpenalty, range }
   struct proximity; { offset, count }
   struct state; { collided, atgoal, orientation, randseed }
};
```

Figure 6: CUDA agent data structure aligned to 16 bytes in global memory; internal data structures are padded to a 16 bytes entity, their members are annotated in green (takes a total of 192 bytes).

Our agent proximity data structure is kept in a separate SoA of $\{ neighbor\_id, neighbor\_type \}$ elements, with neighbor type being either an agent or an obstacle. With no dynamic allocation on the GPU we claim enough space to accommodate the maximum neighbor count specified in the governing simulation parameters, per agent. An agent resorts to its

private { $offset, count$ }, array index state to dereference its own proximity list.

With this data layout, making 8 and 16 bytes our dominant access grain, we appear better positioned to mitigate bandwidth drop of non coalesced transfers. Also, we are less concerned with onetime pointer indirection cost attributed in fetching the index of a thread aligned, array list member; this is well amortized across multiple list element loads or stores.

4.2 Nearest Neighbors Search

One of the key simulator computation steps in arriving at the agent proximity scope is performing k-nearest neighbors search. We found the naïve approach, with each agent iterating over all other agents and computing all possible distances, to pose a quadratic effect on system level running time, resulting in a considerable performance drop on the GPU. Running the exhaustive type search in parallel [Garcia et al. 2008] has merit for higher space dimensionality (>16) and was not an ideal option for us. We therefore leaned towards a spatial hashing scheme that is not necessarily perfect [Lefebvre and Hoppe 2006], but is deterministic in evaluating an upper and lower bound of a query. The hash table assumes as close as linear storage $O(n)$ complexity and performs $O(\log n)$ query time [Overmars 1992], with n the number of agents in the environment. The hash table is recomputed every timestep and therefore must be constructed with a predictable algorithm that takes a relatively small fraction of overall simulation frame time.

The hash function $h(p)$ maps a 3D position $p(x, y, z)$ onto a 1D index for dereferencing the array of agent data structures, discussed earlier. Our mapping simply examines the signed distance of the query position to a reference agent position: $\det(p_{query}, p_{ref})$. Assuming non overlapping agents, an agent position is guaranteed to be unique per frame; however, few hash collisions are unavoidable due to perfect symmetry of agents to a reference agent. Our hash table is built as a balanced, binary tree, representing the static state of agents at the start of a simulation step. And, with a logarithmic query traversal, the prospect for coalesced memory loads is thereby raised. We reload the computed hash table in global memory each frame, and store it as an array of tree nodes, each aligned to a 16 byte boundary. A node data structure is composed of a three dimensional, float data type key, an integer scalar index value, and two nodal device pointers – left and right.

The algorithm for querying the closest agent neighbors takes then the following steps:

For each agent:

- Select random, 3D position samples in the agent radial neighbor area, defined in the global simulation parameters.

For each sample:

- Hash 3D position to get the closest agent index.
- Compute distance between the agent and its closest.
- Insert and sort distance into the agent proximity list.

The number of hash queries we perform per step, for each agent, is externally controlled by the global, nearest neighbors count quantity. Obviously, the neighbor sampling quality directly affects the smoothness of the simulation. In a dense environment we would want to pick a higher sample count to avoid missing close neighbors and as a result agents grazing each other. In practice, a few tens of samples are adequate. The performance benefit of the hashing scheme on the GPU is nonetheless fairly conclusive, once the agent count exceeds a threshold (>500), and further discussed in section 5.

4.3 Execution Model

Launch Overhead. CUDA kernel launch overhead is of prime concern to us. Specifically, we seek minimizing per frame cost incurred by host-to-device and device-to-host copies. The recursive nature of simulation mandates resources to be persistent across kernel launches, both intra and inter frames. While most of our working set is allocated and copied from host-to-device once for the entire simulation session in an insignificant percentage of total running time, the hash table remains a concern. Ultimately, we wanted to run hash table construction on the GPU and avoid unnecessary frame copy. However, launching a single threaded, high latency dedicated kernel incurs an appreciable performance drop and our present implementation resorts to building the table on the CPU followed by a host-to-device frame copy. We address quantitative results on the matter later.

Kernel	Registers	Shared	Local	Constant
simulate	32	116	244	208
update	14	60	0	56

Table 1: Kernel resources; shared, local and constant memory usage in bytes, mostly compiler implicitly generated for kernel launch arguments, synchronization barriers and register spilling.

Property	Kernel	
	simulate	update
Threads Per Block	128	128
Active Threads Per Multiprocessor	512	1024
Active Warps Per Multiprocessor	16	32
Occupancy	50%	100%

Table 2: CUDA static occupancy measure for the simulator kernels, running on a 1.3 compute capable device.

Configuration. Hardware threads are laid out in a single, one dimensional grid of one dimensional thread blocks. A thread is assigned an agent in flat mode, and an agent velocity sample in nested parallel. And, all GPU's currently running threads must complete before any of the kernels is allowed to launch.

Complexity. The simulate kernel performs the three stages of (a) computing a preferred velocity, (b) nearest neighbors search and (c) iterating the new velocity selection out of the combined RVO and VO avoidance set. In theory, the running time complexity of (a) is linear with neighboring roadmap nodes to any of a roadmap node or an agent goal, or with the length of the shortest path from the goal to a roadmap node. Given n the number of agents, k the proximity sample count and v the number of candidate velocities, the algorithm of (b) runs in either $O(n^2)$ without hashing or $O(k \log n)$ with hashing; and (c) performs in $O(vk)$. With the update program executing only a few tens of mostly unconditional device instructions, evidently either (b) or (c) are the de facto performance limiting code sections of our simulator, attracting most of the optimization attention.

Hardware Resources. The GPU results we present in this paper are from running our simulator on NVIDIA's GeForce GTX280

[Nvidia 2008]. Table 1 illustrates hardware resource usage for our kernels. Shared memory usage is primarily static and assigned for launch argument passing and synchronization barriers. Notably high is the simulate kernel register count, 32, imposing a 0.5 thread block efficiency, occupancy, cap as shown in Table 2. Thread local memory area is indeed required for register spilling.

Latency. GTX280 is considered a CUDA 1.3 compute capable device with a total of 16384 registers per multiprocessor. Ideally, we would opt for the upper register bound in assigning warps to a thread block. However, our experiments pointed to an appreciable performance sweet spot in allocating four warps per block, leading to the scheduling of up to four active thread blocks per multiprocessor. With up to 16 active warps per multiprocessor we are able to hide well the latency of a typical fused multiply-add (FMA) instruction (10–12 cycles). Consequently, by effectively processing independent math instructions we are properly positioned to amortize several hundred cycles of global memory access latency.

Coalescing. Providing to CUDA 16 bytes aligned SoA with a small footprint array element, ensures at most two memory load or store instructions per thread access, leading to a worst case bandwidth of a quarter of the fully coalesced half-warp. In addition, we were able to relegate fine grain coalescing to an impressively improved compute capable device 1.2 or higher. With transaction addresses binned into n contiguous segments in memory there could be any number of memory transfers from one to n (up to 16) for the half-warp. Whereas lower compute capable devices are limited to issue either one or sixteen memory transactions. Exceptionally, the new graceful coalescing protocol effectively mitigates poor locality of reference across threads often found in the highly irregular and nested simulate kernel.

4.4 Nested Parallel

With the GPU capable of running simultaneously many thousands of threads in flight, straight forward agent based, flat data parallel often times renders the hardware sub optimally. In fact, simulation scenarios of up to a hundred of agents map on to a single multiprocessor, leaving majority of the thirty available multiprocessors on GTX280 idling. To our knowledge, no work has considered so far velocity level, nested parallel to accomplish improved performance scale for relatively small agent count.

```
 1:  __global__ void
 2:  candidate(CUAgent* agents,
 3:                int index,
 4:                CUNeighbor* neighbors)
 5:  {
 6:     float3 v, float t;
 7:     CUAgent a = agents[index];
 8:
 9:     if(!getThreadId()) v = a.prefvelocity;
10:     else v = velocitySample(a);
11:     t = neighbor(a, agents, neighbors, v);
12:
13:     float p = penalty(a, v, t);
14:     reduceMinAtomicCAS(a, p);
15:     if(p == a.minpenalty)  a.candidate = v;
16:  }
```

Figure 7: Top level, nested candidate velocity, CUDA kernel; performed at its tail, a fine reduce-min into shared memory followed by a coarse global atomic compare-and-swap operation.

The formulation of nested parallel in our simulator forks off a child grid for each agent with its thread count configured by the externally set, number of samples to iterate for resolving the new agent velocity. Nested thread grids run independently given sufficient hardware resources, and they all execute the same candidate velocity, CUDA kernel. Each child grid synchronizes its own threads by performing at the tail of the kernel a fine reduce-min operation into shared memory followed by an inter thread block, global atomic compare-and-swap operation, shown in Figure 7. Of course, in nested mode the original simulate kernel resorts to a lesser compute load. Note that velocity grids inherit resources from the single agent grid, and more importantly the memory footprint remains invariable to flat or nested parallel mode, apart from additional 512 bytes of shared memory.

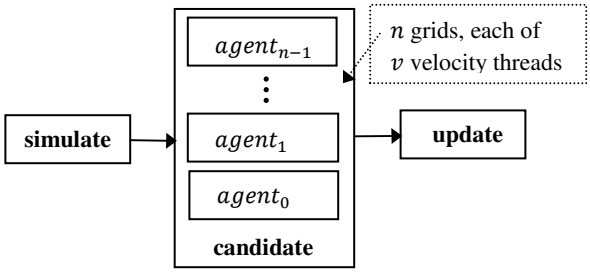

Figure 8: Thread grid distribution in nested simulator pipeline.

The formation of the nested simulator pipeline (Figure 8) introduces thread amplification mid pipe. And, with hundreds of velocity samples to iterate, we can easily populate thousands of threads in the GPU for even a handful of participating agents; thus leveraging the hardware compute power much more effectively.

4.5 Thread Safe RNG

Our pseudo random number generator implements the rand() function of the C runtime library. We realized the generator has to be thread safe and considered to store a seed per multiprocessor in shared memory. This implied a global memory save and a restore per timestep, for persistency. But, updating the seed for every rand() call required an atomic MAD operation into shared memory that turned out to be adversely in appreciably affecting performance. Subsequently, we ended up with a random seed member in the CUDA agent data structure (Figure 6), resulting in an independent random number generator per agent with a more qualitative, well distributed and less predictable number sequence.

5 Results

We next report simulation-only results, all obtained in running on Microsoft 32 bit, Vista and using CUDA 2.1.

5.1 Experiments

Our list of experiments (Table 3) straddles a range of agent count from a handful up to tens of thousands. Simple is a simulation of four agents moving towards a goal diagonally across each other, and bypassing an obstacle in the middle. Second and third respectively, simulate averting collision with a moving obstacle, and robots maneuvering in an area filled with static polygonal obstacles. Next, thirty two agents positioned on a circle move to their diametrically opposite position while yielding the right of way. Then, we simulated four groups of twenty five agents each, forced to pass through a narrow corridor. The Stadium scenario

simulates two hundred and forty five agents entering the field through four gates and then spreading towards forming a textual pattern. Whereas the Crosswalk dataset experiments with four groups of one hundred agents each that establish straight lanes to avoid collisions as they cross each other on a walkway (Figure 14). Finally, we simulated an office floor evacuation with agents escaping the building through two narrow exits (Figure 15). This sequence highlights interesting congestion phenomena that occur with groups travel at different speeds. Leaving the roadmap fixed, the Evacuation setup has multiple representations with ascending number of agents, from five hundreds to twenty thousands. Indeed, for datasets without an explicit definition of a roadmap, agents head on towards their goal from every location reached. Table 4 illustrates governing settings that affect system level, simulation runtime complexity.

Dataset	Agents	Segments	Roadmap Nodes	Flat Thread Blocks
Simple	4	4	4	1
Car	12	0	0	1
Robots	24	130	16	1
Circle	32	0	0	1
Narrow	100	16	0	1
Stadium	245	24	16	2
Crosswalk	400	8	0	4
Evacuation	500	212	429	4
	1000			8
	5000			40
	10000			79
	20000			157

Table 3: Simulation experiments with a quantitative breakdown of simulation objects and flat thread block distribution.

Dataset	Timestep	Proximity		Velocity Samples	Frames
		n	d		
Simple	0.25	10	15	250	607
Car	0.125		100	500	228
Robots	0.1		3	400	455
Circle	0.125		2	500	596
Narrow	0.125		2	500	780
Stadium	0.125		3	500	1116
Crosswalk	0.0625		10	1000	1200
Evacuation†	0.1	10	15	250	1200

†applies to all Evacuation dataset derivatives

Table 4: Governing simulation parameters: timestep in seconds, proximity neighbor count n and radial distance d, candidate velocity samples, and number of actual frames per session.

Property	GTX280	X7350†
Core Clock (MHz)	601	2930
Memory Clock (MHz)	1107	1066
Global Memory (MBytes)	1024	8192
Multiprocessor	30	4
Total Threads	4–20000	16

†Intel's X7350 used in the work by Van Den Berg et al. [2008-I3D]

Table 5: Processor configuration properties for GPU and CPU reported in our results; multiprocessor notation is the equivalent of cores for the CPU.

Table 5 provides configuration properties for the processor types reported in our results (section 5.2). CPU running time on Intel's Xeon X7350 [Intel 2008], for the different evacuation scenarios, are published in the work by Van Den Berg et al. [2008-I3D].

5.2 Statistics

In running the simulations described we were mainly interested in collecting statistic data related to the model memory footprint, parallelism efficiency as simulation progresses, and comparative running time and speedup figures. Figure 9 shows GPU global memory area of simulation resources. Data structures are broken down by association to agents, goals, and the hash table. Roadmap nodes and edges, and at-goal, position and velocity output arrays are rather a small constant and excluded from the chart.

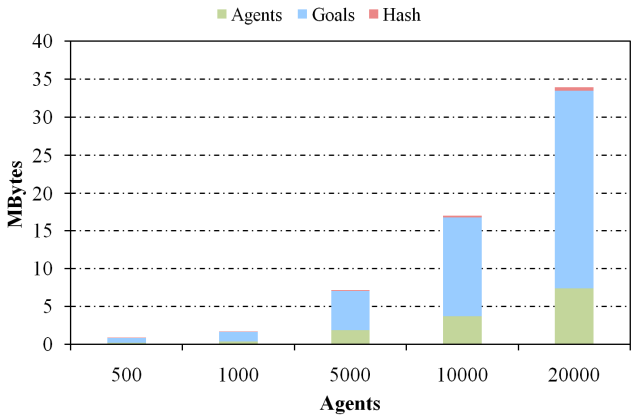

Figure 9: GPU global memory footprint (MBytes) of SoA and AoS entities collectively assembled under agents, goals and hash table subgroups.

The plot in Figure 10 depicts percentage of agents reaching their goal position as a function of simulator advancement in time. A steep agent completion curve towards the end of the simulation implies a more consistent thread block load balance and is seen in the Simple, Circle and Narrow experiments. However, agent progress in the Stadium and the Evacuation plans is more graceful in time and consequently expose thread task unevenness; a block mixing mostly idling with compute intensive threads for at-goal agents and ones still actively pursuing their goal, respectively.

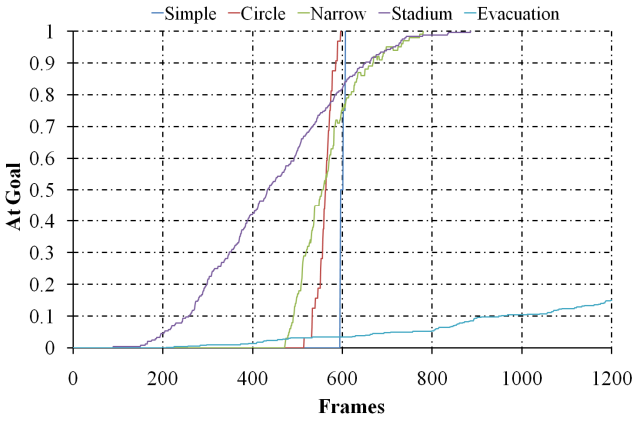

Figure 10: Simulation progress plot for some of our experiments; depicting fraction of agents reaching their goal as a function of frame advancement.

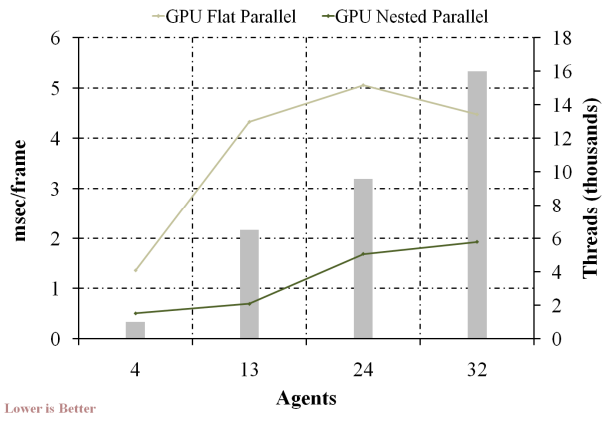

Figure 11: Running time (milliseconds per frame) for simulations with small number of agents (≤32). Comparing the GPU in flat vs. nested parallel modes; thread count shown for nested simulation.

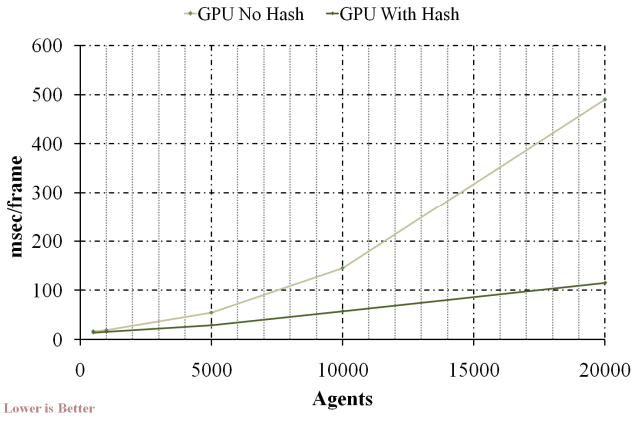

Figure 12: Running time (milliseconds per frame) for simulations with over five hundred agents (Evacuation). Comparing the GPU with and without nearest neighbors hashing.

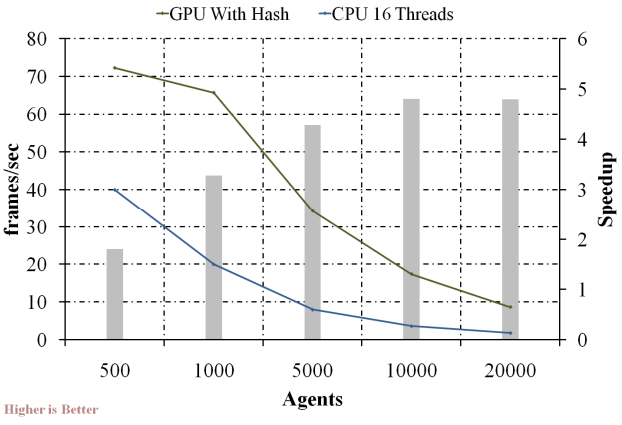

Figure 13: Frame rate (frames per second) and speedup for simulations with over five hundred agents. Comparing GPU with nearest neighbors hashing vs. sixteen threaded CPU.

Figure 11 shows performance of flat vs. nested parallel modes for experiments of small number of agents (≤32). Whereas for simulations with over five hundred agents, Figure 12 summarizes running time for NVIDIA's GTX280 configured with and without hash optimization. And, Figure 13 provides both simulation frame rates and speedup factors of the GPU with nearest neighbors hashing vs. a sixteen threaded RVOLib [RVOLib 2008] invocation on Intel's X7350 CPU. All running times reported on the GPU include both onetime and per frame memcpy from host-to-device and device-to-host, and hash table build time when applicable.

6 Discussion

The statistic data presented in the previous section set forth context to the discussion.

Memory Footprint. The compact working set of our simulator is a striking difference compared to the more resource demanding, continuum perspective frameworks, reviewed in section 2. Furthermore, our video memory footprint is fully deterministic, growing almost linearly with the number of agents. Consequently, we allocate resource space once for the entire simulation session, provided our claimed area fits available GPU global memory, before we launch the simulator. Note that with a narrower global memory stride we are less likely to hit a significant penalty of almost doubling memory access latency (from 400 to 750 cycles) for missing address translation table entries.

Nested Parallel. For small number of agents (≤32), flat level concurrency on the GPU has a diminishing return against the CPU. It is the inner, velocity level parallel that truly brings out the GPU compute power to its fullest, with a realized average of 3.5X acceleration for nested vs. flat parallel modes (Figure 11). Evidently, programming CUDA for nested parallel in our model is remarkably intuitive, but appeared much harder on the CPU given the language tools presently available.

Performance. The hash optimization of section 4.2 improved our simulation performance on the GPU by a factor of up to 4X, compared to the naïve, exhaustive approach (Figure 12). This ultimately affecting the GTX280 to scale almost linearly with increased agent count of up to several tens of thousands. Moreover, we have achieved interactive rate of eighteen frames per second in simulating the Evacuation setup with ten thousand agents (Figure 13). In contrasting our system with a sixteen thread invocation on a 3GHz, quad-core CPU, we have observed a GPU speedup of up to 4.8X. Note that RVOLib would appear benefitting little if at all in optimizing *k*-nearest neighbors search [Van Den Berg et al. 2008-I3D].

Persistent Threading. The reality for memory resources to be persistent has ruled out an extensive use of the GPU shared memory in our design. Nonetheless, managing shared memory as a global memory cache remains a credible optimization option to further reduce overall latency. Specifically, caching frequently accessed agent structure members per thread, could have aided us the most. However, in sustaining four active thread blocks per multiprocessor, for effectively hiding arithmetic operation latency, we are being left with 4 KBytes of shared memory to spare. And, with 128 threads per block, there remain 32 bytes total for agent caching. Restrictive as it stands, we still look forward to experiment with a software controlled cache, expecting next generation hardware to continue relax shared memory space.

Limitations. One constraint in our present design is the single threaded construction of the hash table. Invoking a dedicated GPU kernel for this task would appear the cleaner solution. However, running it single threaded, unable to properly hide global memory latency, is highly incommensurate with observed overhead in the

order of a hash-less, frame time. Whereas, a cache friendly, hash table build on the CPU, combined with the copy to global memory, takes a much lesser toll of 28 percent of update time, for scenarios with tens of thousands agents. Running effectively a single thread, memory bound task on the GPU still remains an implementation challenge.

Hash based nearest neighbors search might impact simulation smoothness and cause agent grazing, once the proximity area is under sampled. Unfortunately, there is no automatic way for picking the 'right' samples. However, conscious oversampling, when the hash function returns a recurred index, mitigates perceived motion artifacts and yet sustains the tremendous performance advantage of hashing.

Lastly, thread amplification in nested parallel can overrun rather quickly both the maximum grid count and in flight threads (15360 on GTX280) the hardware is capable of. We approach this constraint by adaptively selecting flat vs. nested modes based on both the agent and the velocity sample counts.

7 Summary and Future Work

We have extended RVO to perform efficiently on current generation of GPU architectures. Compared to the naïve nearest neighbors search, we found spatial hashing on the GPU to be extremely beneficial, yielding up to quadrupled performance, and leading to almost linear scale of running time with increased number of agents. We addressed severely idling GPU concerns for small agent count by exploiting nested parallel, showing a considerable 3.5X average gain vs. flat mode. On the other hand, we found the GPU notably superior for higher agent count with observed speedup of up to 4.8X compared to a sixteen thread invocation on a powerful, quad-core CPU. Finally, with the advantage of our compact memory footprint we believe the time is appropriate for developers to evaluate, and possibly integrate, GPU based multi agent navigation in game engines.

Areas considered for future work include:

- Develop a consistent simulation framework on the GPU for AI and physics. Per frame interaction involves physics to own the game character position and AI controls its velocity.
- Exploit GPU shared memory as a managed global memory cache to make single threaded, hash table build run effectively.
- Dynamically regroup agents per frame, specifically extracting the ones that reached their goal, to improve thread block load balance.
- Explore split frame simulations on multi GPU configurations; advocate GPU level shared memory to avoid working set replication and expensive inter-GPU copy.
- Improve hash based, nearest neighbors sampling quality by defining finer angular samples and ensuring more consistent area coverage.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive and helpful comments.

References

ABE, Y., and YOSHIKI, M. 2001. Collision Avoidance Method for Multiple Autonomous Mobile Agents by Implicit Cooperation. In *Proceedeings IEEE International Conference on Robotics and Automation*, 1207–1212.

BLEIWEISS, A. 2008. GPU Accelerated Pathfinding. In Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 66–73.

BUCK, I., and HANRAHAN, P. 2003. Data Parallel Computation on Graphics Hardware. *Tech. Report 2003-03, Stanford University Computer Science Department*.

FEURTEY, F. 2000. Simulating the Collision Avoidance Behavior of Pedestrians. *Master's Thesis, University of Tokyo*.

FICKETT, M. W., and ZARKO, L. T. 2007. GPU Continuum Crowds. *CIS Final Project Final report, University of Pennsylvania*.

FIORINI, P., and SHILLER, Z. 1998. Motion Planning in Dynamic Environments using Velocity Obstacles. In *International Journal of Robotics Research,* 760–772.

FULGENZI, C., SPALANZANI, A., and LAUGIER, C. 2007. Dynamic Obstacle Avoidance in Uncertain Environment Combining PVOs and Occupancy Grid. In *Proceedings IEEE International Conference on Robotics and Automation*, 1610–1616.

GARCIA, V., DEBREUVE, E., and BARLAUD, M. 2008. Fast k Nearest Neighbor Search using GPU. In *Proceedings of Computer Vision and Pattern Recognition Workshops*, 1–6.

INTEL, 2007. Intel Core 2 Duo processor. http://www.intel.com/design/core2duo/documentation.htm.

INTEL, 2008. Intel Xeon processor 7000 series. http://www.intel.com/performance/server/xeon_mp/summary.htm.

JEONG, W., and WHITAKER, R. T. 2007. A Fast Eikonal Equation Solver for Parallel Systems. *In SIAM Conference on Computational Science and Engineering*.

KARTHIKEYAN, M. 2008. Real Time Crowd Visualization using the GPU. *Master's Thesis, Virginia Polytechnic and State University*.

KLUGE, B., and PRASSLER, E. 2007. Reflective Navigation: Individual Behaviors and Group Behaviors. In *Proceedings IEEE International Conference on Robotics and Automation*, 4172–4177.

KOVAR, L., GLEICHER, M., and PIGHIN, F. H. 2002. Motion Graphs. *In ACM Transactions on Graphics*, 473–482.

KRISHNA, K. M., and HEXMOOR, H. 2004. Reactive Collision Avoidance of Multiple Moving Agents by Cooperation and Conflict Propagation. In *Proceedings of IEEE International Conference on Robotics and Automation*, 2141–2146.

LaValle, S. M. 2006. Planning Algorithms. Cambridge University Press, http://msl.cs.uiuc.edu/planning/.

Lefebvre, S., and Hoppe, H. 2006. Perfect Spatial Hashing. In *ACM Transactions on Graphics*, 579–588.

Li, Y., and Gupta, K. 2007. Motion Planning of Multiple Agents in Virtual Environments on Parallel Architectures. In *Proceedings IEEE International Conference on Robotics and Automation*, 1009–1014.

Nvidia, 2007. CUDA Programming Guide. http://www.nvidia.com/object/cuda_home.html

Nvidia, 2008. Geforce 200 series: http://www.nvidia.com/object/geforce_gtx_280.html.

Overmars, M. H. 1992. Point Location in Fat Subdivisions. *Information Processing Letters*, 261–265.

Russel, S. J., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*, Prentice Hall, 97–104.

RVOlib, 2008. Reciprocal Velocity Obstacles for Real Time Multi Agent Simulation. http://www.cs.unc.edu/~geom/RVO/Library/index.html.

Shamos, M. I., and Hoey, D. 1976. Geometric Intersection Problems. *17th IEEE Annual Symposium on Foundations of Computer Science*, 208-215.

Shiller, Z., Large, F., and Sekhavat, S. 2001. Motion Planning in Dynamic Environments: Obstacles Moving along Arbitrary Trajectories. In *Proceedings IEEE International Conference on Robotics and Automation*, 3716–3721.

Treuille, A., Cooper, S., and Popovic, Z. 2006. Continuum Crowds. In *ACM Transactions on Graphics*, 1160–1168.

Treuille, A., Lee, Y., and Popović, Z. 2007. Near-Optimal Character Animation with Continuous Control. *ACM Transactions on Graphics*, 26, 3.

Van Den Berg, J., Lin, M., and Manocha, D. 2008. Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation. In *Proceedings IEEE International Conference on Robotics and Automation*.

Van Den Berg, J., Patil, S., Sewall, J., Manocha, D., and Lin M. 2008. Interactive Navigation of Multiple Agents in Crowded Environments. *Symposium on Interactive 3D Graphics and Games*, 139–147.

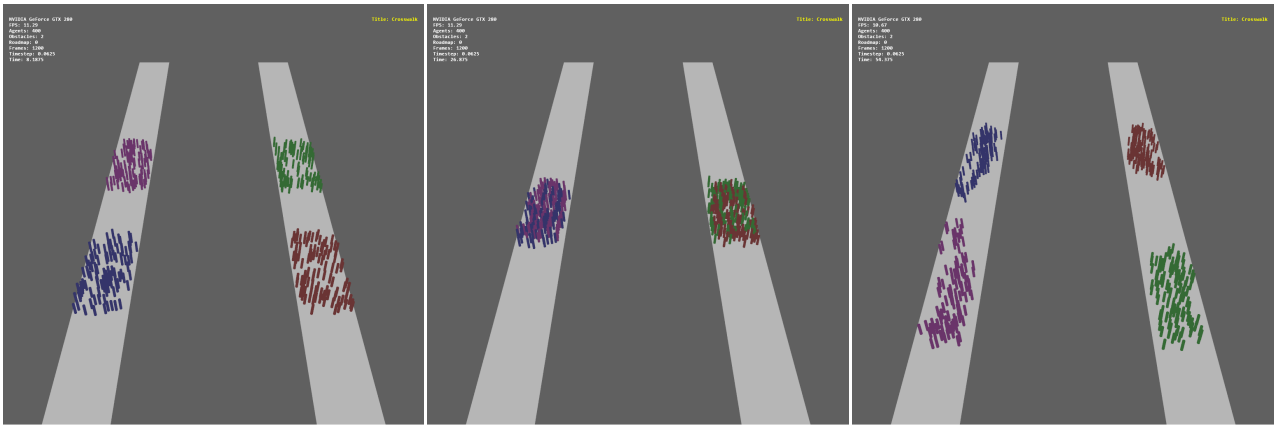

Figure 14: Crosswalk simulation: four groups of one hundred agents each form straight lanes as they cross each other on a walkway.

Figure 15: Evacuation simulation: five hundred agents evacuating an office floor, escaping through two narrow exits; the scenario highlights congestion phenomenon as agents leave the building.