GPU Pathfinding Optimization

Adônis Silva Fernando Rocha Artur Santos
Geber Ramalho Veronica Teichrieb

Federal University of Pernambuco, Center of Informatics - CIn, Brazil

Abstract

In recent years, graphics processing units (GPUs) have shown a significant advance of computational resources available for the use of non-graphical applications. The ability to solve problems involving parallel computing as well as the development of new architectures that supports this new paradigm, such as CUDA, has encouraged the use of GPU for general purpose applications, especially in games. Some parallel tasks which were CPU based are being ported over to the GPU due to theirs superior performance. One of these tasks is the pathfinding of an agent over a game map, which has already achieved a better performance on GPU, but is still limited. This paper describes some optimizations to a GPU pathfinding implementation, addressing a larger work set (agents and nodes) with good performance.

Keywords: pathfinding, intelligent agent, games, CUDA, GPU, A*

Authors' contact:
{ats,fafr,als3,glr,vt}@cin.ufpe.br

1. Introduction

Nowadays entertainment industry moves billions of dollars; however it is not cinema, but games, that stays in the top of the list. Games like *Call of Duty: Modern Warfare 2*, have reached historic levels, such as the amounts generated by the Avatar's film [Humphries 2010]. In order to games moves this quantity of money, there are many factors involved, such as gameplay and graphics realism.

To enhance gameplay and therefore the player's immersion in the game environment, it is necessary to use artificial intelligence (AI) techniques [Mateas 2003]. However, the use of AI techniques results in a higher processing cost, often impairing the performance of the game and forcing, many times, the best AI techniques to be relegated and not used. As a result, the game fails to not have a great acceptance among the players for not presenting challenges with enough level of difficulty [Csikszentmihalyi 2003].

Trying to introduce new experiences, several studies have been performed seeking better performance in order to allow the application of AI techniques to digital games. Many of these techniques are derived from graphical rendering, as Level of Detail - LOD [Sery et al. 2006] and crowds processing techniques [Treuille et al. 2006].

However, some physical limitations such as high frequency, high heat generation in a small area, and electromagnetic interference, resulted in the need for a shift in the architecture of processors, thus allowing the emergence of commercial, multi-core processor [AMD 2005]. Later, the idea of multiple cores has joined other processing devices, like graphics cards.

The architecture of current GPUs allows a high computing power, reaching more than 20x the power of a high-performance processor, as the case of directly comparison between the Intel Core i7 and a GTX400 Series GPU [Owens et al. 2006]. This great performance achieved by the GPUs can even surpass Moore's Law.

The big difference in performance between CPUs and GPUs can be attributed to the differences in their architectures: the CPUs are optimized for high performance on sequential code execution, having thus many sub tasks dedicated to support flow control and cache data, while the GPU processors are designed for parallel processing of instructions, following the concept SIMD (Single Instruction, Multiple Data), thus having more components dedicated to the processing of instructions [Owens et al. 2006]. With this development, approaches have emerged for general purpose processing on GPUs, trying to harness the parallelism of these high power graphics processors.

Conversely, not all applications will achieve a better performance when migrated to the GPU. Features like the degree of parallelism and its model of memory access can allow a better application perform if processed in the CPU. Furthermore, the architecture of graphics cards still has some limitations as the lack of a cache hierarchy, the emergence of double-precision processing only on newer cards, and the existing bottlenecks in data transfer between CPU and GPU.

With the ability to utilize graphics cards for general processing, taking advantage of its inherent parallelism, AI techniques can be processed in a less costly way for the CPU allowing several other parts of the game to be processed without suffering loss of performance. To enable this new programming paradigm, it was necessary that new architectures were developed, as the case of CUDA's architecture (Compute Unified Device Architecture [CUDA 2010]) from NVIDIA.

Thus, this work shows how much performance can be gained by using CUDA to process an A* pathfinding algorithm and how some minor changes, seeking a better use of the architecture prepared by CUDA, can get even better performance gains in processing one of the most basic activities in games, but almost one of the most necessary: calculating the best path between two points, known as pathfinding. In the second section we review the literature about what is being researched on the subject. Later we show how we developed our GPU based implementation and its optimizations, followed by the results comparing this implementation with a CPU one.

In the last and fifth section, we conclude the work and outline the next steps to be done in this research.

2. Related Work

In a game universe, one of the worst challenges is find a way, often the best, between two points that represents the origin and the destiny of a character movement. This problem can be defined as pathfinding. The pathfinding process results in a list of points that represent the character path between the two points.

The path quality and the memory consumption can determine the success of the resulted path [Tozour 2003]. Hence, some decisions about the search space can determine the performance and the effectiveness of the search algorithm. In that way, Tozour shows some possibilities and the influences in the performance results, such as the search space representation in Regular Grids, Navigation Meshes or Waypoint Graphs.

In addition, some games can have thousands of entities that are not controlled by the user, known as non-player character (NPC). Those entities should have their path calculated in a dynamic way, avoiding static and dynamic (the case of others NPCs in the environment) obstacles [LaValle 2006].

Hence, some studies were made seeking better performances, avoiding the decreasing of global game performance. With the advent of CUDA architecture, the game industry started to use it to process the difficult task of pathfinding, increasing the number of researches in this area.

Harish and Narayanan in their study [2007] show the migration of some graph search algorithms from CPU to GPU. They present implementations of the algorithms: **(i)** BFS (Breadth First Search); **(ii)** SSSP (Single Source Shortest Path); and **(iii)** APSP (All Pair Shortest Path). As result, the implementations show similar or faster performance, when compared to the same algorithms processed in a supercomputer that cost five or six times more expensive than the graphic hardware used in the study.

Another work is the one made by Bleiweiss [2008] that adapt the Dijkstra and A* algorithms using the parallelism of GPU. To test if the algorithms had any gains, he tested some roadmaps varying the number of agents and nodes of the graph generated. The focus of his work is not the pathfinding with possible optimizations or with collision detection (techniques that would made the character movement more smoothly, similar to a human movement), but assert the porting of this kind of AI techniques could be ported to GPU and find some improvements that increase the use of the parallel architecture of a graphic card. As results, Bleiweiss achieved a speedup of 24x, in spite of some constraints, as: **(i**) reduced maps (a maximum of 340 nodes – which means a 18x18 navigable map); **(ii)** few agents; and **(iii)** high statically allocated memory consumption.

Reynolds [Reynolds 2006] made his work in a similar way, but using the power of the PlayStation 3® hardware to improve the performance. So, Reynolds partitioned the problem in smaller jobs and each of it is processed by one of the Synergistic Processor Units (SPUs) which composes the console hardware [Pham et al. 2005].

Fisher in the work *GPU Accelerated Path-planning for Multi-agents in Virtual Environments* [Fisher 2009] shows the parallelization of the previous work using the CUDA architecture; shows some modifications to reduce the cost of memory transactions between CPU and GPU; and shows that the implementation using the graphics hardware improved up 56 times when compared with the sequential version.

As shown in this section, most studies have failed to exploit some improvements that are available in the new generations graphic card. Those improvements can increase the performance when compared with a simple sequential version processed in the CPU.

3. Implementation

This work aims to exploit the parallelism in performing the navigation of thousands of agents in a game. Essentially, the goal is to demonstrate the potential of a GPU pathfinding implementation compared to a CPU implementation, showing the speedup acquired. It was thus chosen CUDA architecture for the GPU pathfinding.

CUDA [CUDA 2010] concerns the general purpose parallel architecture developed by NVIDIA in order to fully exploit the advantages offered by graphics cards which are essential in the performance of parallel computing applications. The following sections present an overview of the CUDA architecture followed by a

detailed description of the navigation planning algorithm.

3.1 CUDA

CUDA is a relatively new hardware and software architecture designed to facilitate managing the GPU as a device for general-purpose parallel computing. Using this framework, the graphics card is viewed as a device capable of executing a large number of threads in parallel. Thus, a single program, called a kernel and written in a C extended programming language, which facilitates the development of CUDA based programs, is compiled to the device instruction set and operates on different data elements simultaneously.

The batch of threads that executes a kernel is organized as a grid of thread blocks. In CUDA, threads can be considered the basic units of parallel processing. Each thread on the GPU performs the same function as the kernel and has an ID and local memory. They are organized in blocks and can synchronize its execution and share the same memory space, known as shared memory. A set of blocks represents a grid, which can be one-dimensional or two-dimensional. The grid, in turn, owns a global memory, a constant memory and texture memory that can be accessed by each block which composes it. A kernel consists in the code that is executed on the GPU. For each kernel call, a configuration containing the number of blocks in each grid, the number of threads per block and, optionally, the amount of shared memory to be allocated and the stream associated with the kernel, are needed.

The device thread scheduler decomposes a thread block onto smaller thread groups, usually 32 threads, called warps. Occupancy refers to the ratio between the number of active warps per multiprocessor and the maximum active warps permitted. This concept helps in the understanding of how efficient a kernel could be on the GPU. Having a higher occupancy, usually results in a higher performance. For applications that aren't high arithmetic, such as pathfinding, the peak occupancy reaches 75%. CUDA's Occupancy Calculator [CUDA 2010] further assists to find the best configuration to use all resources offered by the GPUs. The compute capability of the device exploited in the parallel pathfinding implementation presented in this paper complies with CUDA version 1.2. Table 1 shows the output generated for blocks of 384 threads (with 21 registers and 44 bytes of shared memory) used in this work.

Table 1 - CUDA's Occupancy Calculator tool generated output for the pathfinding block of 384 threads

Threads per Block	384
Registers per Block	8192
Warps per Block	12
Threads per Multiprocessor	768
Thread Blocks per Multiprocessor	2

3.2 Basic Implementation

In order to validate the gain achieved by processing the A* pathfinding algorithm on the GPU, a CPU solution was implemented for comparison purposes. Some optimizations with respect to multi-core processors or intrinsic SIMD calls (SSE - Streaming SIMD Extensions) [Bleiweiss 2008] are beyond the scope of this paper. However, some modifications, such as the open node list as a heap based priority queue, proposed by Rabin [2000] were developed to make possible performing a consistent comparison.

Aiming to serve as a basis for understanding the CUDA architecture and to investigate aspects of parallelization presented in agents, we have done an A* pathfinding implementation based on what is proposed in [Bleiweiss 2008]. The relevant aspects of this implementation and subsequent performed optimizations are described in the following sections.

3.2.1 A* using CUDA

Similarly to the CPU implementation, the graph representing the game map is encapsulated in an adjacency lists data structure. However, due to some GPU architecture restrictions, such as memory alignment, texture allocation and use of priority queue, some modifications were done. Furthermore, as the parallelization is referred to the agents, each of them is treated as a thread in a block and executes a complete A* pathfinding.

The graph is divided into three main structures, so that memory remained aligned:

- Nodes: represented by four floats (total of 16 bytes) – one for the id corresponding to the node and three to store the node position (x,y,z) in the world. Despite it offers 3D support, we worked only with a 2D space;
- Edges: they are also represented by four floats – two of them to store its connections (origin, destination), one to store the cost and other reserved just to coalesce the memory;
- Adjacency directory: represents the node's set of edges and is composed of two non-negative integers (total of 8 bytes) – one indicates the offset into the edge list, and the other shows the offset plus the node's count of edges. The use of this adjacency directory, although incurs an extra cost of 8*N bytes compared to an equivalent CPU implementation, contributes to a more efficient navigation.

As the graph created is basically a query structure, it is suitable to remain in a fast access memory region. Therefore, all structures which represent the graph have been mapped to the texture memory in GPU. The main advantage of using textures is that it behaves like a cache, allowing high transfer rates for localized

accesses. It also supports larger graphs since the amount of available texture memory is proportional to the size of GPU RAM.

We also applied some constraints regarding to the agent movement in the map: every node is navigable and can be part of an agent path (i.e. there is no obstacles); the agent can only move in horizontal and vertical, not in diagonal directions, as the map was divided into a grid.

The priority queue implementation, which represents the open nodes of the algorithm, is one of the most important aspects in the A* execution. In the A* main loop, the priority queue is the most accessed structure and impacts directly on the algorithm performance. For this reason, it is restricted to each specific agent and allocated in the local memory, reaching a maximum of 16KB per thread. The priority queue was implemented as a binary heap, previously allocated with size equals to the number of nodes in the graph, while maintaining a set of pairs composed by a float type cost and an integer node id. Elements with the smallest cost are placed on top of the queue. The insert and extract operations, essential to the A* execution, were implemented with a logarithm cost, avoiding recursions. The code bellow lists the heap based extract method.

```
__device__ CUCost
extractFromQueue(CUPriorityQ* pq,
unsigned int* qSize) {
   CUCost cost;
   if((*qSize) >= 1) {
      cost = pq->costs[0];
      pq->costs[0] = pq->costs[(*qSize)-
1];
      (*qSize)--;
      heapify(pq, qSize);
   }
   return cost;
}
```

3.2.2 Working Set

To implement the pathfinding algorithm on the GPU, we defined a working set composed of 4 inputs and 2 outputs. The inputs are each in the form of an array and the described below are:

- An array containing the paths of all agents, represented by a pair (origin, destination);
- An array of costs (float) initialized to zero, representing the cost of each node starting from the initial node;
- Two arrays of integers representing the list of open and closed nodes/edges.

The size of both arrays of costs, open and closed nodes/edges is of A * N, where A is the number of agents in the game and N the number of nodes of the graph. The set of outputs consists of:

- An array of accumulated costs (float), containing the sum of the costs of each path, for each agent;
- An array node positions (float3), containing the paths found for each agent and, in the worst case, having the size of A * N.

This working set is present throughout the execution of the algorithm and is all previously allocated and initialized, so there is no dynamic allocation.

3.2.3 Execution

During the execution, the configuration of the kernel is calculated based on number of agents in the game. Initially, the origin and destination of the agent is set randomly and an A* is performed for this pair in the game map. The software then queries the properties of the graphics device using CUDA to prevent the A* exceeds the available resources of the GPU. Thus, it is estimated the maximum number of agents that may have their ways calculated and then the kernel runs in loop; partial results are then copied to the CPU after execution iteration.

This mechanism of splitting the working set, adapting it to the limit of available memory allows increasing the number of agents, approaching a real world situation, where there are thousands of agents in a game. Because the index of threads can exceed the total number of agents for that kernel call, an initial check is made to disallow the GPU to compute something beyond what should be done.

At the end of the pathfinding execution for each thread, the algorithm outputs the paths found for each agent along with the cumulative cost of the path calculated by A*. During the implementation and development of the navigation algorithm on GPU, it was possible to identify some bottlenecks and optimizations that further increase the gain achieved by the implementation on the graphics cards. These changes can be viewed in the following section.

3.2.4 Optimizations

From the basic pathfinding algorithm implementation on the GPU, it was possible to identify bottlenecks and optimizations that allow a much larger gain compared to CPU implementation and to the algorithm extension supporting a greater tests set, close to what exists in current games – thousands of agents and huge map.

The execution of the kernel loop increases overhead in the existing partial data from the GPU to the CPU, reducing the performance of pathfinding. To minimize the time spent, rather than storing the position of the agent (float3) in the output array that contains the path found, we stored the id of the correspondent node (unsigned int), decreasing the amount of data transferred (float 3 to unsigned int).

Moreover, this same output array was allocated in the CPU non-paged memory, when available, where there is less control by the operating system and consequently an improvement in communication between the video card and RAM memory bus of the motherboard. These modifications improved by 3x the transfer rate of partial results.

Another optimization performed is related to a relatively new concept and was first used in Ray-Tracing by Aila [AILA et. al 2009]. The idea is basically to make the warps more independent and efficient. On GPU, a block execution only ends when all of the running warps finish. In the context of pathfinding, warps within the same block can have very different paths. Thus, at least one of the warps would spend more time calculating the path than another one, leaving the block, somehow inefficient. With the addition of persistent threads [AILA et. al 2009], each warp works almost independently and does not expect the end of the other's execution to start, increasing the performance of the algorithm. Despite the overhead this technique brings, we implemented the persistent threads using atomic operations in the shared memory instead of global memory, because of the fast access of shared memory.

For a better understanding of CUDA architecture, we have also made some modifications: heuristic calculation (Euclidean distance) without using the `sqrt` function, which is an expensive operation to CUDA and a tuning to reduce the number of registers used and the transfer of data to global memory.

Based on these optimizations performed, it was possible to see gains in comparison to the basic implementation on the GPU, strengthening even more the idea of porting the pathfinding to use the computing power of graphics hardware. The results and their implications are detailed in the following section.

4. Results

The A* pathfinding algorithms implemented in CPU and GPU and described in section 3 are analyzed and compared in this section. Evaluated from the perspective of performance gain (speedup) and memory consumption, this analysis had significant results, making possible seeing that the parallelization, at the level of agents and for the covered algorithms, showed some speedup, strengthening the idea of using the GPU as a platform for general purpose applications, with emphasis on adapting algorithms of Artificial Intelligence in Games.

We used undirected graphs for benchmark and generated them automatically with a low complexity topology. The number of agents used was, in average, the square of nodes quantity, excepted for some benchmarks that focused on running the algorithm with a fairly large number of agents and nodes. The paths of each agent were randomly generated and we used a capability of 1.2, compatible with all graphics cards currently on the market. All results described in this section performed on an Intel Core i7 1.6GHz with 4GB of RAM processor, for both CPU and GPU implementations and an NVIDIA GeForce GTS 360M with 1782MHz shader clock, 1GB of global memory and 12 multiprocessors graphics card. The speedup was measured comparing the single-threaded CPU algorithm with the ones implemented on GPU. The pathfinding ran on Windows 7 and the time was measured using the timer of Windows API.

As the paths were generated randomly, each benchmark ran three times and the average between the time measured and memory used was taken to provide a more consistent result. Thus, we performed a comparison between the performance obtained with implementations in CPU, basic GPU and GPU optimized, with and without persistent threads. In Table 2 you can see all benchmarks used in the tests.

Table 2 - Benchmark lists; with the nodes and edges numbers, quantity of agents (threads), the number of blocks without persistence (384 threads per block)

Graph	Nodes	Edges	Agents	Blocks
G0	16	48	64	1
G1	64	224	1024	3
G2	144	528	20736	54
G3	256	960	65536	171
G4	324	1224	115600	302
G5	400	1520	300000	784
G6	900	3480	20736	54
G7	2025	7920	1024	3
G8	2025	7920	65536	150

From the values obtained by the average, we calculated the speedup that the GPU in comparison to the CPU. The speedup achieved for each benchmark can be seen in Figure 1. It is observed that the gain is greater as the complexity of the map increases (larger number of nodes) and the number of agents as well, reaching its peak in G3. We can also see that for a very small set, as in G0, the CPU implementation still represents a good solution. On the other hand, with a very large working set, the GPU implementation achieves a speedup of about 6x.

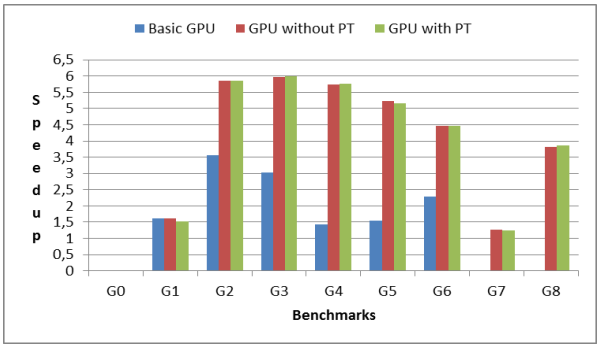

Figure 1 - Comparison of A* performance of basic GPU, with and without persistent threads (PT) vs. the CPU implementation.

Although the speedup obtained is less than in [Bleiweiss 2008], the main contribution of this work is the number of agents and map size we achieved. While in [Bleiweiss 2008] is used a map with maximum of 340 nodes, we achieved a maximum of 2025 nodes with 65536 agents, and a maximum of 300000 agents with a game map of 400 nodes, which represents a set closer to a real game application. As the memory of the graphics cards is still limited, making a pathfinding implementation with a high scalability, close to real world, is a common challenge and growing research area. In Figure 2 is also possible to view the execution time of each A* implementation done which shows that the execution time grows with the topology of the game map. Observing the execution time of the G8 set on CPU, we can see that, with a large working set, which demands a lot of processing power from CPU, the GPU implementation represents a great solution.

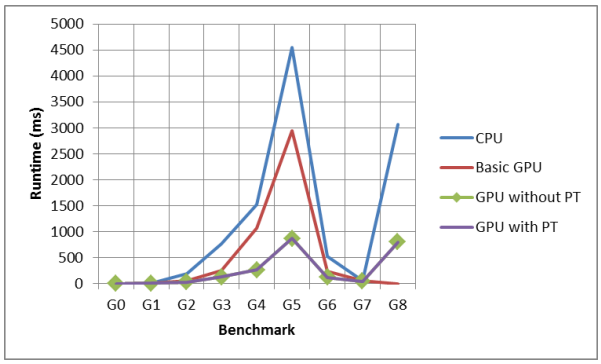

Figure 2 – Absolute execution time of A* implementation on CPU, basic GPU and GPU with and without persistent threads (PT).

5. Conclusion

This paper has presented a wide study of the techniques related to the solution of the agent terrain navigation problem, seeking its direct application in game development. The main objective of this work was to develop a pathfinding algorithm on GPU and figure possible and effective optimizations, taking advantage of the potential of parallel graphics processors and the CUDA architecture, allowing the use of these multiprocessors in the problem of navigation.

For this reason, it is possible to visualize the potential of GPUs in the pathfinding execution. The current games have the tendency of bringing huge and complex environments, with simulations of thousands of agents in real time. With the limitations imposed by the CPU architecture and resources, some AI techniques are having performance problems to execute in CPUs. Along with the constant and fast evolution of computing power of graphics processors, the implementation of those AI techniques, mainly those with a high parallelism degree, became very promising in GPUs, establishing a bridge to a possible future in games.

As future work, some improvements and modifications are listed below:

- Reduce the working set, mainly using dynamic allocation;
- Expansion to multiple GPUs, noting the impact of replication map of the game;
- Investigate the possibility of multi-agent approaches, where an agent can reuse the previously calculated by another way;
- Investigate the possibility of another approach to parallelization, differently from the one presented in this work – one agent, one thread.

This paper presents a solution to the problem of navigation of agents using the GPU as a development platform. We intend to improve the implementation of the A* algorithm so that it can use all the resources and computing power of graphics hardware. In addition, there is the possibility of establishing a benchmark that can serve as a basis for similar applications to be tested, since there is no standardization of the sets of tests, parameters, etc. Thus, we understand that it would be possible to evaluate the performance of different navigation algorithms in a consistent and effective way on the GPU.

References

AILA, T., LAINE, S., 2009. *Understanding the Efficiency of Ray Traversal on GPUs. In: Proceedings of the Conference on High Performance Graphics 2009 (HPG 2009)*, 145–149.

AMD, 2005. *Multi-core processors the next evolution in computing*. Amd multi-core technology whitepaper.

BLEIWEISS, A., 2008. GPU Accelerated Pathfinding. In Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS *Symposium on Graphics Hardware*, Aire-la-Ville. Eurographics Association, 65-74.

CUDA, 2010. *Computed Unified Device Architecture* [online]. Available from: www.nvidia.com/ object/cuda_home_new.html [Accessed 20th July 2010].

CSIKSZENTMIHALYI, M., 1991. Flow: The Psychology of Optimal Experience. Harper Perennial.

FISHER, L. G., SILVEIRA, R. NEDEL, L., 2009. GPU Accelerated Path-planning for Multi-agents in Virtual Environments. *In SBGames*, 2009.

HARISH, P. AND NARAYANAN, P. J., 2007. Accelerating large graph algorithms on the GPU using CUDA. *In: Proceedings of the International Conference on High Performance Computing* (18-21 December 2007 Goa). Springer-Verlag, 197-208.

HUMPHRIES, J., 2010. Avatar Vs Modern Warfare 2: The billion dollar behemoths [online] Business Management. Available from: www.bme.eu.com/news/avatar-vs-modern-warfare-2/ [Acessed 27th July 2010].

LAVALLE, S. M., 2006. Planning Algorithms [online] Cambridge University Press. Available from: http://planning.cs.uiuc.edu/ [Acessed 20th July 2010]

MATEAS, M., 2003. Expressive AI: Games and Artificial Intelligence. *In: Proceedings of International DiGRA Conference.*

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E. AND PURCELL, T. J., 2005. A Survey of General-Purpose Computation on Graphics Hardware. *In Start of The Art Report of the Eurographics Conference.*

PHAM, D., et al (20 authors). 2005. The Design and Implementation of a First-Generation CELL Processor. In *Solid-State Circuit Conference, 2005. ISSCC 2005 IEEE International*.

RABIN, S. 2000. A* Speed Optimizations. *Game Programming Gems*. Charles River Media, 2000. 272–287.

REYNOLDS, C. 2006. Big fast crowds on ps3. *In sandbox'06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, ACM Press, New York, NY, USA, 113–121.

SERÝ, O., POCH, T., SAFRATA, P. AND BROM, C., 2006. Level-of-detail in behaviour of virtual humans. *In Proceeding of the Conference on Current Trends in Theory and Practice of Computer Science*, 21-27 January 2006 Merín. Heidelberg: Springer-Verlag, 565–574.

TOZOUR, P., 2003. Search Space Representations. *In AI Game Programming Wisdom 2*, editors, Charles River Media, pages 405-415. Hingham, Massachusetts.

TREUILLE, A. COOPER, S. AND POPOVIC, Z., 2006. Continuum Crowds. *In: Proceedings of SIGGRAPH 2006*. ACM Trans. Graph. 25(3).