ISSN 2007-9737

Simulating and Visualizing Real-Time Crowds on GPU Clusters

Benjamín Hernández[1], Hugo Pérez[1,2], Isaac Rudomin[1], Sergio Ruiz[3], Oriam de Gyves[4], and Leonel Toledo[4]

[1]Barcelona Supercomputing Center, Barcelona,
Spain

[2]Universitat Politècnica de Catalunya, Barcelona,
Spain

[3]Tecnológico de Monterrey, Campus Ciudad de México,
Mexico

[4]Tecnológico de Monterrey, Campus Estado de México,
Mexico

{benjamin.hernandez, hugo.perez, isaac.rudomin}@bsc.es,
{sergio.ruiz, odegyves, ltoledo}@itesm.mx

Abstract. We present a set of algorithms for simulating and visualizing real-time crowds in GPU (Graphics Processing Units) clusters. First we present crowd simulation and rendering techniques that take advantage of single GPU machines. Then, using as an example a wandering crowd behavior simulation algorithm, we explain how this kind of algorithms can be extended for their use in GPU cluster environments. We also present a visualization architecture that renders the simulation results using detailed 3D virtual characters. This architecture is adaptable in order to support the Barcelona Supercomputing Center (BSC) infrastructure. The results show that our algorithms are scalable in different hardware platforms including embedded systems, desktop GPUs, and GPU clusters, in particular, the BSC's Minotauro cluster.

Keywords. Crowd simulation, visualization, HPC, GPU-clusters, real-time, embedded systems.

1 Introduction

Crowd simulations allow safe application of the scientific method to certain subsets of the crowd phenomena; they may aid in the analysis of different events related, for example, disease propagation, building evacuations, traffic modeling, or social evolution. The use of computational models that describe the behavior of people is becoming a necessity as the population increases in urban areas. Prediction before, during, and after daily crowd events may reduce associated logistic costs.

Large scale crowd simulations demand computational power and memory resources commonly available in HPC (High Performance Computing) platforms, they particularly require in situ visualization, i.e., when significant computational power for concurrent execution of simulation and visualization is required. Coupling visualization with simulation while it is running reduces bottlenecks associated with storing, retrieving, and post processing data in disk storage.

The aim of this paper is to show how an algorithm implementing a wandering crowd behavior (a simple behavior that is commonly used in crowd simulations) can be extended for its use in GPU (Graphics Processing Unit) cluster environments. In addition, we present an adaptable visualization architecture that renders in 3D the simulation results and supports three configurations: streaming, in situ, and web.

2 Related Work

Reynolds [11] proposed the first known simulation solution for large groups of entities with emergent behavior being an extension of particle systems. It

doi: 10.13053/CyS-18-4-2060

is based on three basic rules: separation, alignment, and cohesion. These rules keep together, while going in a given direction and free of collisions, a group of boids or bird-like objects. Such model is an example of an individual-based or agent-based model (ABM), and is used to simulate the global behavior of a large number of interacting autonomous individuals. ABMs can be applied to different fields including biology, ecology, and economics.

ABMs applied to crowd simulation require the processing of each individual separately, thus it is a good candidate for data-parallel processing. In recent years, simulation systems have exploited the computing power offered by the GPU, freeing the CPU from tasks that are highly parallel and based on a single instruction multiple data (SIMD) model. Before the existence of programming models such as OpenCL or CUDA, GPUs were already used for general purpose computing. For example, Rudomin et al. [14] used Finite State Machines to determine the behavior of agents, which are implemented using fragment shaders in GLSL (OpenGL Shading Language).

CUDA and OpenCL allowed developers to take advantage of the GPU resources using a more traditional C-language syntax. In addition, the architecture of the GPU has evolved to meet the demands of general computing as well as that of graphics. In this sense, Yilmaz et al. [21] proposed a fuzzy logic method implemented in CUDA for simulation of crowds.

There is a large body of work in simulation and visualization of crowds. Some of it uses the GPU. Since much of this is beyond the scope of this article, for a more comprehensive and recent discussion, consult Rudomin et al. [13] and Kappadia et al. [5]. Both articles discuss several advanced methods for simulating, generating, animating, and rendering crowds.

Computer clusters have also been used for crowd simulation and visualization. For example, Vigueras et al. [19] describe an architecture in which they distribute virtual world regions between different machines called Action Servers (AS), where each of them controls the actions of agents located in their region. Agents are assigned to a Client Process (CP) and maintain constant communication with their AS. The server also maintains communication with AS of adjacent regions to query the state of the world in the border areas. This system is complemented with Visual Client Process or VCP for visualization. There can be several VCP in the system to cover different regions or subregions of the virtual world; the AS sends only the information of the agents that will be displayed by the VCP, which may receive information from one or more AS. The initial algorithms were implemented in CPU; however, in recent versions of the architecture, collision detection process has been migrated to the GPU accelerating the process considerably [18].

If one desires to have crowd simulations that take advantage of the GPU or an HPC cluster without needing to program from scratch, there are a couple of frameworks available where a user can define the characteristics of the simulation and agents through configuration files usually in Extensible Markup Language (XML) format. The most popular is FLAME GPU [12], which maps formal descriptions of agents into simulation code that runs in the GPU. Another example is Pandora [20]. It is an open-source framework designed to create and execute large-scale social simulations in HPC environments. Pandora will automatically deal with the details of distributing the computational load of the simulation, so the researcher does not need to have any additional knowledge about parallel or distributed systems.

3 Single GPU Crowd Simulation and Rendering

Interactive simulation and visualization of large and varied crowds for single GPU systems is a very active research topic. Current trends are focused on semi-automatic crowd authoring (modeling and animation), microscopic and macroscopic behavior, and rendering. As solutions to these challenges, we have developed methods for simulating crowds of varied aspect and a diversity of behaviors.

In this section we present solutions for real-time crowd simulation and rendering that take advantage of single GPU systems.

3.1 Accurate Psycho-physical Characteristics

Crowds are heterogeneous groups of people, in which each person has individual and social properties, and interacts with others through different means. It is not realistic to have a crowd composed of only assertive male pedestrians around 30 years old; nevertheless, simulations do not typically include the characteristics that make pedestrians unique. We focus on the navigation of virtual agents based on accurate psycho-physical characteristics of a population. Individual properties include physical and psychological characteristics; social interaction properties include communication and group formation. Participants from a perception study are able to identify the psychological characteristics of the agents in a simulation and experimental results show that agents with these characteristics also change the behavior of a crowd.

We can define *physical characteristics* as a person's body defining traits which are visually perceptible and generally can be measured. Age, gender, height, weight, and fitness are examples of these characteristics. These physical characteristics are based on studies of real pedestrians [6, 2, 7, 10] and are used to compute the maximum speed at which a virtual agent is able to walk at any given time.

We use the term *psychological characteristics* to refer to the personality of a pedestrian. Personality can be understood as a set of internal motivations that make two people behave in different manners, even if they share similar physical features. We use the Eysenck 3-factor model, which categorizes the personality of an individual according to three main factors: Psychoticism, Extraversion, and Neuroticism. An individual may tend to exhibit one of these factors because of testosterone, serotonin, and dopamine, respectively. Durupinar et al. [1] and Guy et al. [3] studied similar approaches in regard to the implications of adding psychological characteristics to virtual agents. Figure 1 shows a comparison of the three personalities that are achieved using this method.

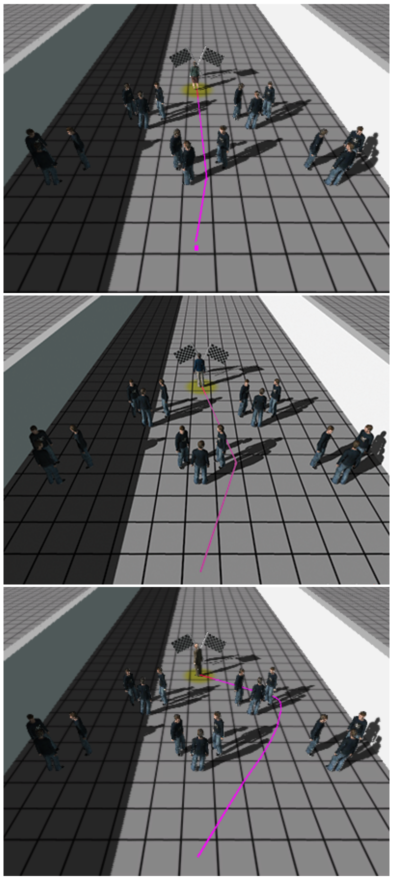


Fig. 1. Psychological comparison. The agent moves through five groups in order to get to its destination. From top to bottom: Aggressive agent, Assertive agent, and Shy agent

3.2 Goal-Oriented Behavior

We now focus on a particular technique for solving the Navigation problem involving embodied agents within virtual environments. Our first observation is that an agent, while moving through an environment, solves a sequential decision problem to find a path that goes from its current configuration to a goal, constructing a set of additive rewards as it gets closer to its goal. This behavior can be described by a Markov Decision Process [15].

Markov Decision Processes (MDP) may be solved in stages by a single GPU in real time, given that the partition of the navigable space is coarse so that the MDP represents general paths for groups of agents [15]. We assign multiple rewards and penalties to cells in a partitioned navigable space to control agent Navigation. A finer grid partition (Figure 2) allows for another algorithm to handle Local Collision Avoidance (LCA).

The LCA algorithm displaces agents toward their goal in the direction established by the MDP Optimal Policy (Π^*). This technique presents three simulation advantages:

Evacuation scenarios. Several exits and obstacles can be modeled, matching real scenarios for evacuation. *Scenario zonal hierarchy.* Zones of different importance can be modeled, simulating rough terrain or agent preference to navigate. *Interaction.* Since the MDP solution is segmented, obstacles or exits can be introduced or removed without the need to stop the simulation.

3.3 Hierarchical Structures for Level of Detail for Varied Animated Crowds

We introduce a level of detail system useful for varied animated crowds, which is capable of handling several thousands of varied animated crowds at interactive frame rates. This is accomplished by using two complementary structures to reduce memory consumption and optimize the rendering stage. The first structure is a skeleton with associated octrees which are used for computing geometry and animation level of detail. The second structure is a tiling mechanism and a quad-tree built on top of this tiling that is used for further level of detail optimization, allowing us to combine geometry from different characters in parts of the scene that are far away from the camera. The combination of these structures allows us to render several thousands of varied characters within a crowd, for instance, crowds up to a quarter million characters are achievable at interactive frame rates using these structures. Extra details on the technique can be found in Toledo et al. [17].

4 Cluster Crowd Simulation

4.1 Algorithm Overview

In this section we present a simplified algorithm to simulate reactive behavior, i.e., crowds of agents

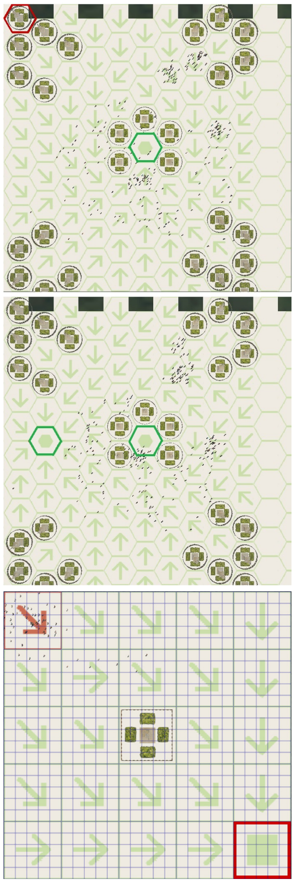


Fig. 2. Navigation and LCA. Top: MDP partition with one exit. A coarse partition provides Navigation for agents. Navigation is modeled with a MDP. Middle: MDP partition with two exits. Several rewards and penalties can be modeled. Bottom: LCA partition. Another parallel algorithm handles LCA using the MDP policy

wandering in a virtual environment while avoiding collisions between each other. This is a conceptually simple behavior; however, in non-optimal conditions, its complexity is $O(N^2)$ (where N is the number of agents in the crowd), i.e., an agent position must be compared with all the remaining positions to detect collisions. Current techniques reduce the complexity of the algorithm using geometrical or hierarchical approaches [5, 13]. We reduce the complexity of the algorithm by partitioning the navigable space into a grid which cuts down the search space. In addition, by defining a search radius for each agent, this search space is reduced even more. This approach follows [11], however, we increase the performance of the algorithm by adopting a parallel programming model based on GPU and multiple nodes (Sec. 4.2).

The algorithm was implemented in CUDA and uses three data arrays that store id, agent, and world information. The id array stores a unique identifier for each agent. The agent array stores the x,y position in the world, the motion's direction (stored as an angle), and the speed at which the agent is moving. The world array stores zeros for unoccupied cells and the agent id for occupied cells.

The agents are initially placed in the world with random direction and speed at random locations. Then the simulation works as follows. Each agent is moved in the direction and with the speed that is contained in its representation, and does so unless this would take the agent to a cell that is occupied. Collision avoidance is calculated by evaluating a radius of five positions around their current position; then the agent will move in the direction in which there are fewer agents or unoccupied cells within such radius. Such evaluation starts using the original agent's direction and switches counterclockwise every 45 degrees covering a total of eight directions (Figure 3 left).

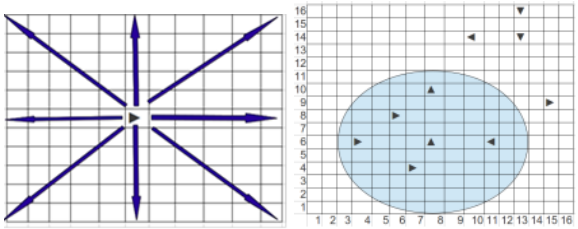


Fig. 3. Left: Evaluation radius and directions for collision avoidance. Right: Path evaluation example

Notice that if there is another agent in the evaluated path having the same direction as the current agent, such cell is considered as a free cell, since it would be no collision. For example, Figure 3 Right shows the case of the agent in the position (8,6). The first direction evaluated is upwards because it is its original direction; after four cells there is another agent but it is going in the same direction, therefore, the cell is considered free. There are other directions where we would also find empty cells in a radius of five positions, for example, in the down direction, but we give priority to the agent's original direction. In the current implementation, agents choose the best cell at every iteration, but the original angle for the agent is not changed, so in the next iteration agents will try to move in the original direction. When an agent reaches the limit of the world, it rotates 180 degrees; for example, the agent at position (15,9) would return in the same path.

The main computation is updating the status of the world and the agents, and this task is performed in the GPU using CUDA. Notice that the data arrays are uploaded to the GPU memory at the beginning of the simulation. On the other hand, by defining a search radius and direction for each agent, our algorithm reduces the original complexity of collision avoidance from $O(N^2)$ to $O(N \times r \times d)$ worst case, where N is the number of agents, $r << N$ is the search radius and $d << N$ is the search direction. The best case occurs when the agent finds another agent just in front of him, i.e., $r = 1$ and $d = 1$ which turns into $O(N)$ complexity, and the worst case occurs when the agent is rounded up by other agents.

4.1.1 Results

Three experiments were designed to verify the performance of the algorithm in different platforms consisting in an embedded system and four different GPUs. The embedded system was a Jetson TK1 development kit with 192 CUDA cores and 1.8 GB VRAM, and the GPUs were a GT 540M with 96 CUDA cores and 2 GB VRAM, a Tesla M2090 with 512 CUDA cores and 6 GB in RAM, a GTX TITAN Black with 2880 CUDA cores and 6 GB VRAM, and a Tesla K40c with 2880 CUDA cores and 12 GB VRAM.

All tests were performed running 30 iterations and using a world size of 16384×16384. Reported time corresponds to the average time per simulation step given in seconds.

The first experiment consisted in determining the speedup of the simulation when GPU processing is present. The sequential version of the algorithm, running in one CPU core of an Intel Xeon E5649

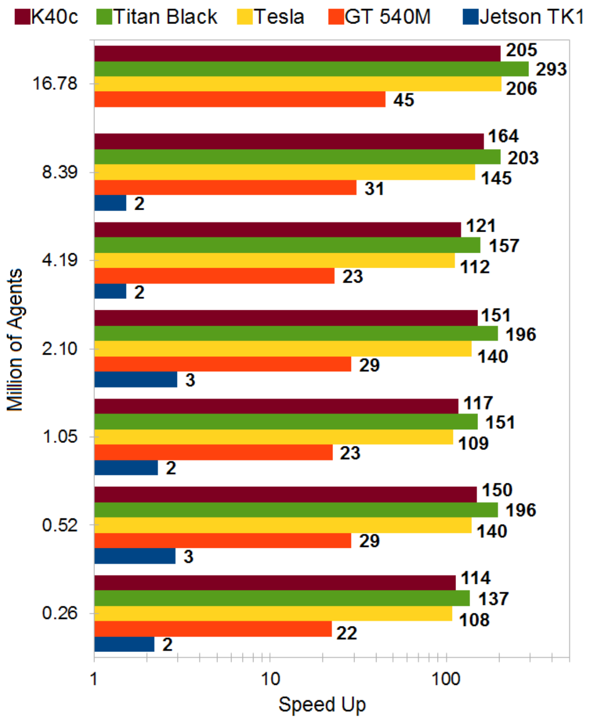


Fig. 4. Speedup obtained in the CUDA version of the algorithm

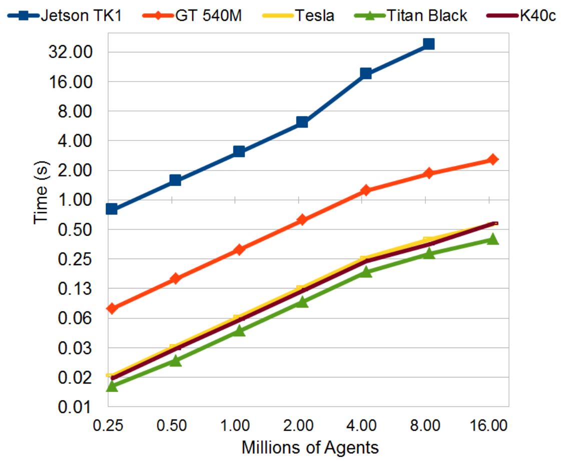


Fig. 5. Performance comparison in different GPUs

Six-Core at 2.53 GHz with 24 GB RAM, is used as a reference. Results of this test are reported in Figure 4.

The second test consisted in evaluating the algorithm's performance on the different hardware platforms described earlier. Figure 5 shows the results from this test. Considering $30ms$ being the threshold response time for interactive systems, according to the results the Jetson TK1 is not able to respond with the minimal quantity of agents which was 512^2 in this test, GT 540M can simulate around 500 thousand agents, Tesla, Titan Black, and K40c can simulate almost a million of agents.

Notice that in these experiments, we simulate up to 16 million agents because most of the GPUs were able to allocate such amount of data except the Jetson TK1 which supported up to eight million agents.

The third experiment allowed us to establish the performance of each platform when reaching its peak load (Figure 6).

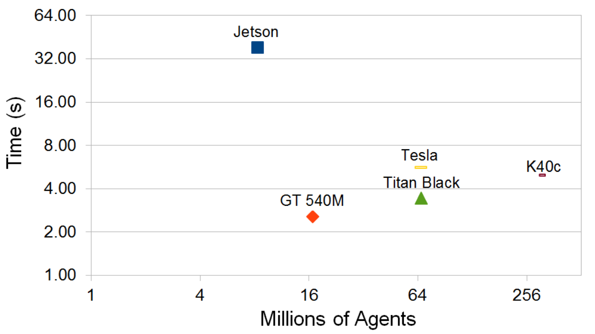


Fig. 6. Peak load of the embedded system and GPUs used in the experiments

4.2 Simulating Crowds in Minotauro Cluster using MPI and CUDA

As found in the experiments in Section 4.1.1, the number of agents to compute in one node is limited by the memory available in the GPU. In particular, this limit is 6 GB for the Minotauro cluster (Nvidia M2090 Tesla GPU); thus, to tackle larger problems we will need to use multiple nodes.

Multiple node programming requires memory management, communication, and synchronization techniques to avoid communication overheads; the system architecture becomes complex

doi: 10.13053/CyS-18-4-2060

because of the processes inherent to the programming model.

In order to extend the algorithm from Section 4.1 to multiple nodes, we implemented the tiling technique, i.e., the world is tiled, then each tile is computed by a different node. In this case, CUDA is used for computing the agent's new position, and MPI (Message Passing Interface) is used for data interchange between nodes.

We designed the system to use a number of nodes that has a square root. The use of the number of nodes that has a square root allows us to divide the world into an equal number of rows and columns (Figure 7). The minimum number of nodes that our program can use to execute the simulation is four. On the other hand, the world can be any size, but large sizes are to be expected given a large number of agents which can be simulated.

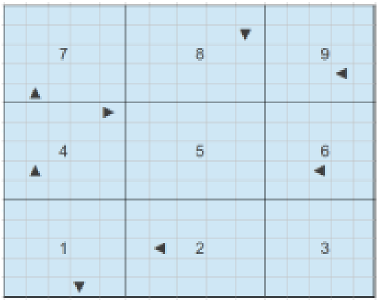


Fig. 7. Example of a simulation requiring 9 nodes: the world is tiled into 9 zones, i.e., 3 rows and 3 columns

Regarding the data structures, we modify the id array to store the zone in which each agent is located. Notice that, to save memory, each node stores just the tile of the world that corresponds to it, using an offset in axis x and y to calculate the global coordinate.

On the other hand, the collision avoidance algorithm is modified when the agents are on the borders of the tile. We explain this with an example. Considering a vision radius of one cell, Figure 8 shows that the agent located at coordinates (10, 9) in zone 5 needs to know if the cell at coordinates (11, 9) is empty: it is necessary to exchange the information of the agents near the borders in order to determine the next position of such agents.

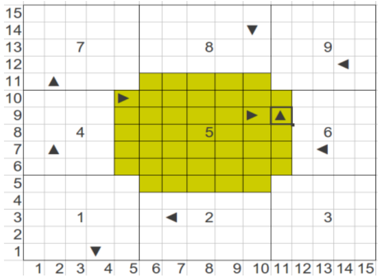


Fig. 8. Interchange of agent information in area bordering two nodes

Therefore, the area considered as the tile's border is the radius times the height of the area for left and right neighbors, and the radius times the width of the area in the case of up and down neighbors.

Communications are made more efficient by exchanging only the cells that are occupied at the borders using a two-dimensional integer array that stores the coordinate of a cell in the world and the identifier of the agent occupying the cell. On the other hand, as computation is performed by several nodes, we are reducing the complexity of the algorithm presented in Section 4.1 to $O(\frac{N \times r \times d}{n})$, where N, r, and d remain as described in Section 4.1, and n is the number of compute nodes. However, this complexity only takes into account an ideal case when nodes do not interchange data. Considering we are using point-to-point communication between n nodes and sending messages of m size, then, the complexity is $O(\frac{N \times r \times d}{n}) + O(nm)$ or $O(\frac{N \times r \times d}{n} + nm)$, with $n << N$ and $m << N$. A worst case for the communication complexity may occur when a node communicates with its four neighbors, e.g., node five in Figure 7, and an example of a best case is node three, which only interchanges data with nodes six and two (Figure 7).

4.2.1 Results

Two experiments were performed in order to verify the efficiency of the algorithm using multiple nodes. In both experiments, the world size was 27852^2 and we ran three iterations of the algorithm.

The first experiment consisted in a performance comparison between the CPU and GPU (CUDA) version of the algorithm using 4, 9, and 16 nodes

(Figure 9). Although the GPU version is better than the CPU version as expected, the difference is not as big as in single node. The GPU version execution time is increased dramatically due to the numerous internode communications, and, in particular, because in order to transfer data between GPUs we must copy data from the GPU memory to the CPU memory, then transfer data between nodes, and then copy from the CPU memory to the GPU memory. The execution time of the CPU version has improved compared to the single node sequential version.

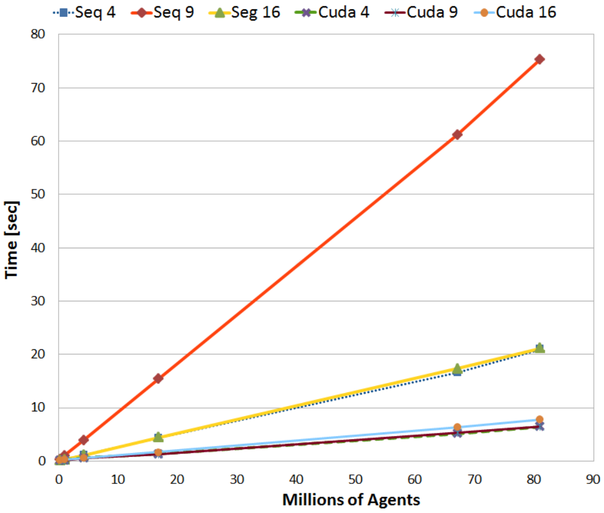


Fig. 9. Performance comparison between CPU and GPU versions using 4, 9, and 16 nodes

The second test was to determine simulation scaling using multiple nodes (Figure 10). In the current implementation it is possible to simulate up to 289 million agents with 4 nodes and 529 million agents with 9 nodes.

The response time for the CUDA multiple node version increases up to 5 seconds for 64 million agents, which is a good number considering the number of agents. On the other hand, the CPU multiple node version improves considerably with respect to the single node version, but the GPU multiple node version is still at least 3 times faster.

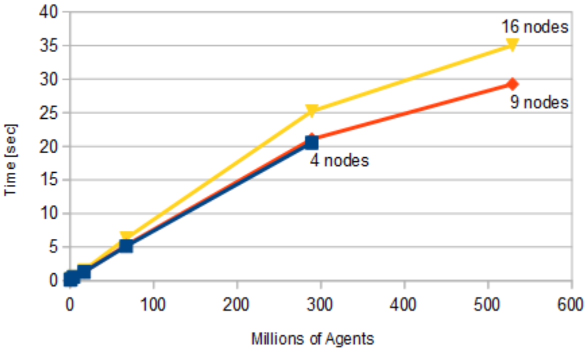


Fig. 10. Scaling

5 GPU Cluster Crowd Visualization

Recent advances in hardware and software technologies in HPC technology have allowed the simulation of large scale problems. Computational solutions produce a vast amount of data which require further processing to offer insights about the results. Such information usually is simplified before analysis; however, simplifications may hide relevant details.

On the other hand, visualization, which transforms these results into graphical and color representations, can improve human cognition in data or results' analysis. In addition, in situ visualization reduces I/O operations [8, 22], i.e., it avoids full data transfer of results; it also reduces time, allowing the researcher to inspect partial simulation results and perform simulation adaptations as required.

The selection of a visualization mechanism depends on the simulation and available HPC infrastructure; our approach for visualization can render detailed 3D crowds. The term 'detailed' refers to the fact that the crowd is rendered using animated characters with different geometrical and visual appearance. In addition, we have developed three visualization modes described in the following paragraphs that are suitable for the particular BSC's infrastructure.

Figure 11 shows the crowd engine's modules designed to visualize detailed 3D crowds. The GoD, or generation of diversity stage, runs as a preprocess. This stage generates and animates 3D characters semi-automatically. In run-time simulation results are stored in an array of ids and agent

doi: 10.13053/CyS-18-4-2060

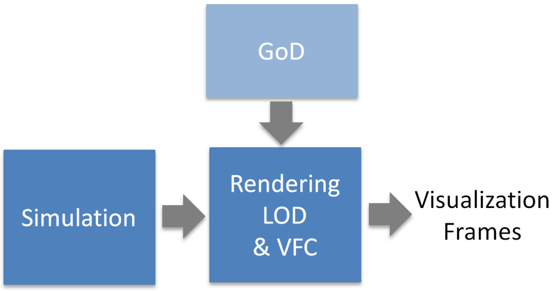


Fig. 11. General diagram of our Crowd Engine. GoD (generation of diversity) generates and animates geometrical and visually diverse characters, then they are rendered using level of detail (LoD) and view frustum culling (VFC) according to the simulation results

positions (Section 4) which are used to render unique animated characters. Details of this stage are reported in [16]. Discrete level of detail (LoD) and view frustum culling (VFC) are implemented in GPU to visualize several hundreds of characters in real time [4]. As a result, our engine generates frames that are processed or transmitted according to different engine configurations.

5.1 Streaming Mode

As mentioned earlier, visualization mechanisms depend on the simulation and available HPC infrastructure. In the case of the Barcelona Supercomputing Center, the Minotauro cluster establishes fast network connection within BSC's LAN, this allows visualization at interactive rates. Default set-up uses VirtualGL[1] for image transport. Bandwidth between the user and the cluster can help or severely impair an interactive user experience. Unfortunately, external bandwidth into BSC is restricted resulting in a visualization frame rate of about 1-2 fps with a resolution of 512^2 pixels using VirtualGL's jpeg compression.

Accessing simulation results for visualization in any computer is one of our goals. It is desirable to enable researchers to access their results for visualization and analysis at a distance, and such access may occur during or after simulation. A second goal is to couple our crowd engine to existing BSC Pandora framework [20] without major modifications in any of the platforms.

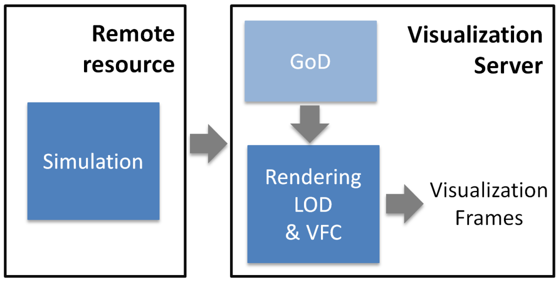


Fig. 12. Our crowd engine can be configured in the streaming mode to visualize simulations or results being calculated or stored in remote resources

In order to achieve both goals, we decided to execute the simulation (Section 4) or access the simulation data from a remote source and stream out the results to a visualization server or a workstation. Figure 12 shows a general diagram of this approach. A remote resource sends crowd simulation results (previously generated or calculated in run-time) to a visualization server which does the crowd rendering.

For the second case, which requires remote resource access, a HDF5 file format parser was designed to retrieve the data. For test purposes OSC protocol is used to stream results in both cases.

Notice that the bandwidth of the connection needs to be less than that needed for transferring images. It depends on the number of agents and not the size or quality of the image. It works for fewer agents but works for slower networks and gives us good quality results.

5.2 In Situ Mode

In Situ Visualization configuration uses the algorithm described in Section 4.2 for simulation using the Minotauro GPU cluster. Each node executes both the simulation and the visualization of one tile of the world (Figure 13). The crowd engine was modified to support MPI communications and off-screen rendering.

[1] http://www.virtualgl.org/

doi: 10.13053/CyS-18-4-2060

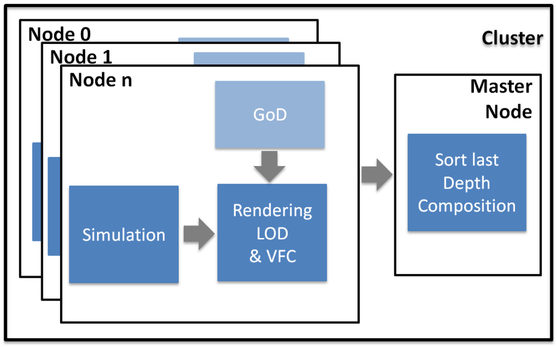


Fig. 13. In situ configuration allows crowd simulation and visualization within different nodes. Each participating node sends its rendered frames to the master node. The master node composes each received frame to generate the final image

MPI communications enable the engine to assign simulation/rendering work to each participating node, share simulation results between nodes (Section 4) and transmit partial renderings for final composition. Off-screen rendering enables each node to capture a visualization frame with color and depth information encoded in an RGBA array where the depth component is stored in the alpha channel. Then this information is downloaded to RAM and transferred to the master node.

Once the master node receives the color plus depth images from each node, it generates the final composite. This image is generated based on sort-last depth compositing algorithm [9] implemented in GLSL. Finally, the composite image can be sent to the client through VirtualGL.

Notice that the client will not require advanced rendering resources as needed in the streaming mode but it needs to be connected to BSC's LAN.

5.3 Web Mode

A particular requirement of our streaming architecture (Section 5.1) is the use of a workstation or visualization server having a last generation GPU. On the other hand, in situ visualization requires direct connection between the user and the cluster.

We are designing a third option in order to fill the gap between streaming and in situ mode. The web mode, currently in the prototype stage, is designed to allow the researcher to access visualization results in any device and away from the desktop.

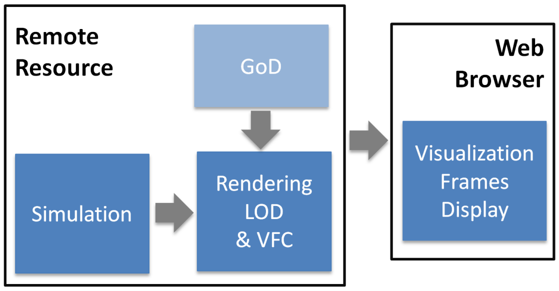


Fig. 14. Web mode allows to display visualization results in web browsers

Inspired by cloud gaming technology, this configuration captures the results from visualization and streams them out to a web browser (Figure 14). It follows the client/server architecture: the server executes the simulation, visualization, and streaming, and the client displays the visualization frames and captures user interaction events which are sent back to the server.

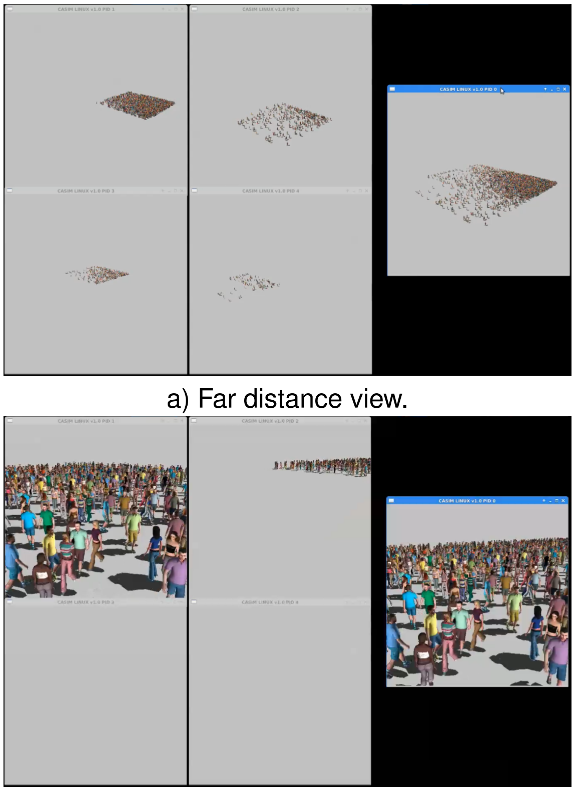
5.4 Results

Stream mode tests were performed using two options. In the first option, a remote resource streams a previously generated simulation in the Pandora Framework of a wandering crowd behavior. The remote resource uses a peer to peer connection to a workstation with an Nvidia TITAN graphics card. In this case we obtained a consistent frame rate of 60 fps for a crowd of 4096 characters.

The second test consisted in setting up a reverse tunneling connection between the Minotauro cluster and the workstation. The simulation is executed in the cluster and results are streamed out to the workstation. In this case we were also able to render 4096 characters at 60 fps.

It is important to mention that our streaming algorithm does not implement any buffering technique. However, our crowd engine interpolates the received character's positions between frames to avoid artifacts due to possible data loss.

In the case of in situ visualization, the test was performed in the Minotauro cluster using five MPI process. Four processes simulated and rendered



a) Far distance view.

b) Close up view.

Fig. 15. In situ mode results. Four nodes are used, each renders the crowd according to the assigned world tile

4096 characters and the fifth one performed image composition (Figure 15 Up). Figure 15 Bottom shows a close up view of the simulation. In the case of in situ visualization, we obtained frame rates between 15 to 20 fps for a simulation of 16K characters.

Further tests consisting in running the crowd engine in one node (Nvidia Tesla M2090) showed that it can render up to 8K characters at 10-12 fps while the workstation with an Nvidia TITAN graphic card can render up to 32K characters at 16-20 fps.

Finally, to test the web prototype, we designed two experiments. The first one consisted in running the prototype in the same machine using different web browsers to detect potential compatibility problems (Figure 16). Our results showed that Chrome, Firefox, and Opera browsers were able to display the visualization successfully. In addition, there were found no noticeable lag between the user interaction and the simulation.

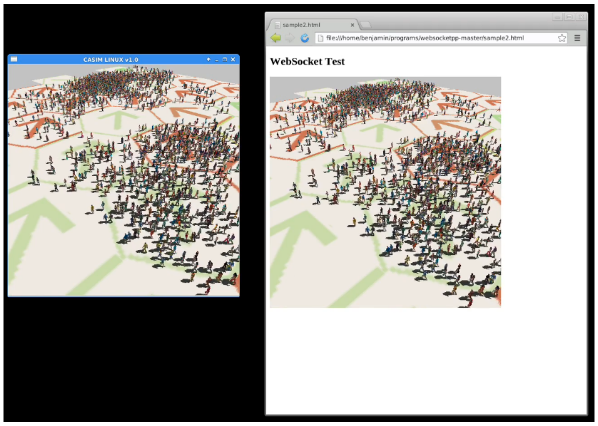

CASIM LINUX v1.0
sample2.html
file:///home/benjamin/programs/websocketpp-master/sample2.html
WebSocket Test

Fig. 16. Results of the web mode prototype. Left: OpenGL/glut window. Right: Visualization is displayed in Chrome browser

In the second experiment, we made a peer to peer connection between the workstation and a laptop, and between the workstation and an Android based mobile device. Some lag between the simulation and the visualization in the browser was noticeable. This is due to the fact that in the current state of the prototype we have not yet implemented any compression mechanism; however, this configuration allows us to inspect visualization results in any device.

6 Conclusions

In the embedded and single GPUs systems, our algorithm scales almost linearly according to its algorithmic complexity. It reaches up to 302 million agents in a single GPU system. A result that attracted our attention was the performance of the Jetson TK1. Despite having 192 CUDA cores, the Jetson TK1 had a poorer performance than the GT 540M which has 96 CUDA cores. This is because the Jetson TK1 has a clock rate of 852 MHz and a memory bus width of 64 bits, while the GT 540 has a clock rate of 1344 MHz and a memory bus width of 128 bits. Nevertheless, the Jetson TK1 performance is up to three times better than that of the sequential version. It is also important to notice that the architecture of the GPUs allows better speedups when processing larger numbers of agents.

The performance behavior of the other GPUs were as expected.

On the other hand, the cluster version of the algorithm scales up to 529 million agents with 9 nodes. It is important to mention that communication overhead is reduced by computation and communication overlapping and exchanging only occupied positions in the borders between neighbors. We also found that scalability is worse for the GPU accelerated application than that for the CPU application given that the impact of the GPU acceleration was quickly dominated by the communication time, since in each iteration we exchange agents and border areas with neighbor nodes. We are working to improve the communication processes.

Regarding visualization, we have designed a configurable crowd engine that supports three basic modes: streaming, in situ, and web. The streaming mechanism supports crowd visualization in machines connected outside the BSC's LAN; in addition, the web mode allows us to display large scale simulations in any device. Both configurations offer flexibility and may reduce infrastructure costs when advanced visualization systems such as the CAVE or Tiled Display Walls are remotely available.

Streaming mode also avoids the difficulty of coupling already existing simulation tools, and the web mode can be used to take advantage of sensors available in mobile devices. On the other hand, our in situ visualization approach for crowd rendering is designed to take advantage of the GPUs available in the Minotauro supercomputer. But additional performance is expected when the simulation runs in one GPU and crowd rendering in the second GPU is available per each node.

7 Future Work

As future work, we plan to use recent techniques of communication like CUDA-aware MPI or GPU Direct, expecting these technologies to improve the communication process. On the other hand, the dynamic nature of crowd simulation makes it prone to load imbalance quickly, so we are adopting OMPSS, a programming model developed at the Barcelona Supercomputing Center, to tackle this problem.

The visualization architecture also needs some improvements. For example, the streaming mode can be used to couple the rendering engine with additional crowd simulation frameworks; the web mode requires mechanisms for adaptive compression and buffering. In addition, an interesting opportunity that the web configuration offers is the use of sensors available in mobile devices (e.g., touch screen, accelerometers, and gyroscopes) for user interaction. A mobile device can also be used as a remote control in cases where visualization is displayed in CAVE systems or Tiled Display Walls.

Finally, detailed rendering demands more resources than simulation. In the current in situ architecture, the amount of simulated agents depends on how many of them can be rendered; in other words, we do not simulate more agents than those that can be visualized per node. Alternative architectures for in situ visualization will use some nodes for simulation and others for visualization. Another important modification that we are considering is the reduction of communications between slave−master nodes: instead of performing the final composition in the master, partial compositions can be done in nodes selected based on the virtual camera. Then each partial composite can be transmitted to the nodes near to the camera to perform additional compositions and, finally, the master will do the global composition of the remaining partial composites. This modification will be necessary for exascale systems which would be composed of thousands of compute nodes.

Acknowledgements

The authors would like to thank NVIDIA through the CUDA Center of Excellence program for partial support of this project. This work has also been partially funded by CONACyT-BSC postdoctoral fellowship, CONACyT SNI-54067, CONACyT CVU-375247, and CONACyT CVU-311633.

References

1. **Durupinar, F., Allbeck, J., Pelechano, N., & Badler, N. I.** (**2008**). Creating Crowd Variation with the OCEAN Personality Model. *International joint conference on Autonomous agents and multiagent systems*, AAMAS, pp. 1217–1220.
2. **Fugger, T., Randles, B., Stein, A., Whiting, W., & Gallagher, B.** (**2000**). Analysis of Pedestrian Gait and Perception-Reaction at Signal-Controlled Crosswalk Intersections. *Transportation Research Record*, Vol. 1705, No. 1, pp. 20–25.
3. **Guy, S. J., Kim, S., Lin, M. C., & Manocha, D.** (**2011**). Simulating heterogeneous crowd behaviors using personality trait theory. *Symposium on Computer Animation*, pp. 43.
4. **Hernández, B. & Rudomin, I.** (**2011**). *A rendering pipeline for real time crowds*. GPU PRO 2. AK Peters.
5. **Kappadia, M., Pelechano, N., Guy, S., Allbeck, J., & Chrysanthou, Y.** (**2014**). Simulating heterogeneous crowds with interactive behaviors. *EG 2014 - Tutorials*.
6. **Knoblauch, R., Pietrucha, M., & Nitzburg, M.** (**1996**). Field Studies of Pedestrian Walking Speed and Start-Up Time. *Transportation Research Record*, Vol. 1538, No. 1, pp. 27–38.
7. **Laplante, J. N. & Kaeser, T. P.** (**2004**). The Continuing Evolution of Pedestrian Walking Speed Assumptions. *ITE Journal*, Vol. 74, No. 9, pp. 32–40.
8. **Ma, K.-L.** (**2009**). In situ visualization at extreme scale: Challenges and opportunities. *Computer Graphics and Applications, IEEE*, Vol. 29, No. 6, pp. 14–19.
9. **Molnar, S., Cox, M., Ellsworth, D., & Fuchs, H.** (**1994**). A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, Vol. 14, No. 4, pp. 23–32.
10. **Rahman, K., Ghani, N. A., Kamil, A. A., & Mustafa, A.** (**2012**). Analysis of Pedestrian Free Flow Walking Speed in a Least Developing Country : A Factorial Design Study. *Research Journal of Applied Sciences, Engineering & Technology*, Vol. 4, No. 21, pp. 4299–4304.
11. **Reynolds, C. W.** (**1987**). Flocks, herds and schools: A distributed behavioral model. *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, ACM, New York, NY, USA, pp. 25–34.
12. **Richmond, P. & Romano, D.** (**2011**). Template-Driven Agent-Based Modeling and Simulation with CUDA. In *GPU Computing Gems Emerald Edition*, Applications of GPU Computing Series, chapter 21. Morgan Kaufmann, 1 edition, 313–324.
13. **Rudomin, I., Hernández, B., de Gyves, O., Toledo, L., Rivalcoba, I., & Ruiz, S.** (**2013**). Gpu generation of large varied animated crowds. *Computación y Sistemas (CyS) special issue on Supercomputing: Applications and Technologies*, Vol. 17, No. 3.
14. **Rudomín, I., Millán, E., & Hernández, B.** (**2005**). Fragment shaders for agent animation using finite state machines. *Simulation Modelling Practice and Theory*, Vol. 13, No. 8, pp. 741–751.
15. **Ruiz, S. & Hernández, B.** (**2014**). Markov decision process and micro scenarios for crowd navigation and collision avoidance. *Research in Computing Science*, Vol. 74, pp. 103–116.
16. **Ruiz, S., Hernández, B., Alvarado, A., & Rudomín, I.** (**2013**). Reducing memory requirements for diverse animated crowds. *Proceedings of Motion on Games*, MIG '13, ACM, New York, NY, USA, pp. 55:77–55:86.
17. **Toledo, L., De Gyves, O., & Rudomín, I.** (**2014**). Hierarchical level of detail for varied animated crowds. *The Visual Computer*, Vol. 30, No. 6-8, pp. 949–961.
18. **Vigueras, G., Orduña, J. M., Lozano, M., & Cecilia, J. M.** (**2014**). Accelerating collision detection for large-scale crowd simulation on multi-core and many-core architectures. *Int. J. High Perform. Comput. Appl.*, Vol. 28, No. 1, pp. 33–49.
19. **Vigueras, G., Orduña, J. M., Lozano, M., & Jégou, Y.** (**2013**). A scalable multiagent system architecture for interactive applications. *Sci. Comput. Program.*, Vol. 78, No. 6, pp. 715–724.
20. **Wittek, P. & Rubio-Campillo, X.** (**2012**). Scalable agent-based modelling with cloud hpc resources for social simulations. *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, pp. 355–362.
21. **Yilmaz, E., Isler, V., & Cetin, Y. Y.** (**2009**). The virtual marathon: Parallel computing supports crowd simulations. *IEEE Computer Graphics and Applications*, Vol. 29, No. 4, pp. 26–33.
22. **Yu, H., Wang, C., Grout, R., Chen, J., & Ma, K.-L.** (**2010**). In situ visualization for large scale combustion simulations. *Computer Graphics and Applications*, Vol. 30, No. 3, pp. 45–57.

Benjamín Hernández is a postdoctoral researcher at the Barcelona Supercomputing Center in Spain. Dr. Hernández focuses his research interests on the intersection of real-time crowd simulation, human computer interaction, and visualization of inhabited virtual environments using high performance computing. He has advised postgraduate theses in this field and co-directed the NVIDIA CUDA Teaching Center Initiative at Tecnologico de Monterrey. He currently holds the National System of Researches Fellowship (SNI-C) from the Mexican National Council for Science and Technology(CONACYT), Mexico.

Hugo Pérez is a Ph.D. student specializing in Computer Science at Universitat Politècnica de Catalunya, Barcelona, Spain. His research interests are real-time crowd simulation, high-performance computing, parallel programming models, and computer graphics.

Isaac Rudomin received his Ph.D. from the University of Pennsylvania. He is currently a senior researcher at the Barcelona Supercomputing Center, Spain. His research interests are human and crowd generation, simulation, animation and visualization, human-computer interaction, and high performance computing.

Sergio Ruiz is a software engineer at the Monterrey Institute of Technology, Mexico City campus (Tecnologico de Monterrey, campus Ciudad de Mexico). Currently, he is a Ph.D. student at the same institution; his research area is path planning for simulated crowds proposing a thesis entitled “A Hybrid Method for Macro and Micro Simulation of Crowd Behavior”. He is currently on a research interchange visit working on crowd simulation at the Barcelona Supercomputing Center, Spain.

Oriam de Gyves is a Ph.D. specializing in Computer Science and particularly in Computer Graphics, at Tecnologico de Monterrey, campus Estado de Mexico. He studies and does research of behaviors for crowd simulation using General Purpose Computation in Graphics Processing Units. His research interests include simulation, behavior and visualization of real-time crowds, as well as parallel computing on GPUs.

Leonel Toledo is a Ph.D. student at Tecnologico de Monterrey, Campus Estado de Mexico. He made a research interchange visit working on crowd simulation at the Barcelona Supercomputing Center, Spain. He has been a half-time professor at Tecnologico de Monterrey, Campus Estado de Mexico, since 2011, and his research interests include crowd simulation, animation, visualization, and rendering.

Article received on 23/06/2014; accepted on 28/09/2014.