BehaveRT: A GPU-Based Library for Autonomous Characters

Ugo Erra[1], Bernardino Frola[2], and Vittorio Scarano[2]

[1] Università della Basilicata, Potenza, Italy
ugo.erra@unibas.it
[2] Università di Salerno, Salerno, Italy
{frola,vitsca}@dia.unisa.it

Abstract. In this work, we present a GPU-based library, called BehaveRT, for the definition, real-time simulation, and visualization of large communities of individuals. We implemented a modular flexible and extensible architecture based on a plug-in infrastructure that enables the creation of a *behavior engine* system core. We used Compute Unified Device Architecture to perform parallel programming and specific memory optimization techniques to exploit the computational power of commodity graphics hardware, enabling developers to focus on the design and implementation of behavioral models. This paper illustrates the architecture of BehaveRT, the core plug-ins, and some case studies. In particular, we show two high-level behavioral models, picture and shape flocking, that generate images and shapes in 3D space by coordinating the positions and color-coding of individuals. We, then, present an environment discretization case study of the interaction of a community with generic virtual scenes such as irregular terrains and buildings.

1 Introduction

Autonomous character behavior simulation in computer graphics is a relatively new and complex area of research that is at the heart of modeling and simulation and behavioral and cognitive psychology. Within the last three decades, researchers in this area have attempted to model behaviors using simulation and visualization primarily for education and training systems. Today, behavior representation is also used in entertainment, commercials and non-educational simulations. The field of application ranges from crowd control [19] and evacuation planning [20] to traffic density [8] and safety [3].

The main goal is the simulation of autonomous virtual characters or agents to generate crowds and other flock-like coordinated group motion. In this type of simulation an agent is assigned a local behavioral model and moves by coordinating with the motion of other agents. The number of agents involved in such collective motion can be huge, from flocks of several hundred birds to schools of millions of fish. These simulations are attracting a large amount of interest [21] because, with appropriate modifications to interaction terms, they could

R. Boulic, Y. Chrysantou, and T. Komura (Eds.): MIG 2010, LNCS 6459, pp. 194–205, 2010.
© Springer-Verlag Berlin Heidelberg 2010

be used for large number of systems, besides human crowds [1], in which local interactions among mobile elements scale to collective behavior, from cell aggregates, including tumors and bacterial bio-film, to aggregates of vertebrates such as migrating wildebeest. For this reason, we also use in this paper the term "individual" to indicate an element in the group.

Today, Graphics processing units (GPUs) are used not only for 3D graphics rendering but also in general-purpose computing because of increases in their price/performance ratio and hardware programmability and because of their huge computational power and speed. In particular, developments in programmability have significantly improved general accessibility thanks to a new generation of high-level languages simplifying the writing of a GPU program. In this scenario, the NVIDIA Compute Unified Device Architecture (CUDA) abstracts GPU as a general-purpose multithreaded SIMD (single instruction, multiple data) architectural model and offers a C-like interface for developers. However, developing an efficient GPU program remains challenging because of the complexity of this new architecture, which involves in-depth knowledge of resources available in terms of thread and memory hierarchy. Such knowledge remains a key factor in exploiting the tremendous computing power of the GPU, although manual optimizations are made time consuming, and it is difficult to attain a significant speed-up [15]. For this reason, several efforts have been devoted to exploit the processing power of GPUs without having to learn the programming tools required to use them directly. For instance, several framework based on the GPU can provides acceleration for the game physics simulation [6][11] and recently the GPU has been also exploited in artificial intelligence technology for path planning [2]. These works show that the level of interest in GPU-accelerated frameworks in video games is growing, and manufacturers are pushing new technologies in the GPUs.

In this work, we present BehaveRT as a flexible and extensible GPU-based library for autonomous characters. The library is written for CUDA language and enables the developer/modeler to implement behavioral models by using a set of plug-ins; the developer can also quickly prototype a new steering behavior by writing new plug-ins. The advantages of our framework are: (i) modelers can focus on specifying behavior and running simulations without in-depth understanding of CUDA's explicit memory hierarchy and GPU optimization strategies; (ii) the simulation performance exploits the computational power of the GPUs and allows massive simulation with high performance; (iii) CPU and GPU memory allocation of the data structures associated with agent population is hidden by BehaveRT, and these structures can be visualized in real time since agent data is already located on the GPU hardware.

The remainder of this paper is organized as follows: in section 2 we review previous relevant frameworks based on CPU and GPU approaches for agent-based simulation. In section 3, we describe the overall system. In section 4, we present the set of core plug-ins. Section 5, illustrates some case studies implemented using BehaveRT. Finally, section 6 concludes and discusses future work.

2 Related Work

Recently, researchers have begun to investigate the parallelism of GPUs in speeding up the simulation of large crowds. In this scenario, GPUs are used essentially for speeding up neighbors searches, which is an essential operation in agent-based simulation. Several approaches have been proposed. In [4], the authors use a GPU that enables the massive simulation and rendering of a behavioral model of a population of about 130 thousand at 50 frames per second. Similarly, the authors in [14] describe an efficient GPU implementation for real-time simulation and visualization. Their framework, called ABGPU, enables massive individual-based modeling underlying the graphical concepts of the GPU. In each case, these works focus more on exploiting GPU architecture for efficient implementation and less on employing a reusable software library.

OpenSteer is an open-source C++ library that implements artificial intelligence steering behaviors for virtual characters [13] for real-time 3D applications. Agents in OpenSteer react to their surroundings only by means of steering behaviors. Several such behaviors are implemented in OpenSteer, such as "walk along a path" or "align with a neighbor", corresponding to a steering vector computed from the current agent's state (position, orientation, velocity) and its local environment (nearby obstacles and neighboring agents, etc.). These simple behaviors may be combined into complex ones by combining the corresponding vectors. Some OpenSteer ideas have been adopted in BehaveRT, and we supply a plug-in with an interface similar to OpenSteer's.

SteerSuite is a flexible framework inspired by OpenSteerDemo component of Reynolds software. It provides functionality related to the testing, evaluating, recording, and infrastructure surrounding a steering algorithm [17].

3 Building a Behavior Engine

BehaveRT is a C++/CUDA library that enables the collective behaviors of a community of individuals by defining a customized *behavior engine*. Individuals belonging to the same community have the same characteristics, behave similarly and can interact with each other. In this paper, the term "behavior" is used to refer to the improvised actions of individuals.

The behavior engine is an entity containing *features* and *functionalities*. Features are data structures containing a number of elements equal to the number of individuals and representing their state, e.g., their position, orientation, velocity and so on. Functionalities are operations applied in a block to the whole community and can modify the features of individuals through behavior modification or can be used to manipulate features for particular operations such as determining all individuals that fall inside a range.

A behavior engine is assembled by means of an ordered set of plug-ins. The set of features and functionalities of the behavior engine is determined by those of each plug-in. Figure 1 shows an example of a behavior engine. The resulting behavior engine is able to manage a community in which each individual has a 3D

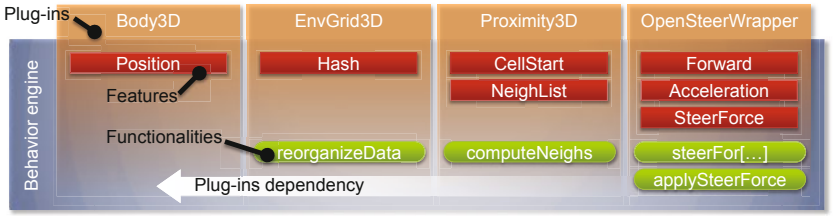

Fig. 1. Plug-in mixin. Body3D adds to the behavior engine one feature and no functionalities. EnvGrid3D adds one feature and a functionality for data reordering. Proximity3D covers related to neighborhood searching, such as neighborhood lists. The plug-in OpenSteerWrapper provides a set of functionalities that implements some steering behaviors. Each plug-in depends on the plug-in(s) to its left. For example, OpenSteerWrapper relies on the features and functionalities of Proximity3D, EnvGrid3D and Body3D.

position, recognize its neighboring individuals, and execute steering behaviors. These four plug-ins define the basic system core for the behavioral modeling of a community of individuals that can interact with each other.

3.1 Functionalities Implementation

Functionalities envelope one or more CUDA kernel calls. BehaveRT simplifies three different implementation phases (Figure 2 shows a comparison between a classical and BehaveRT approach). (*i*) Allocation: features are encapsulated and allocated simultaneously on both CPU and GPU. The library output is bind as OpenGL renderable object. (*ii*) Call: kernels call is simplified thanks to data structures encapsulation and allocation setups. (*iii*) Execution: kernels definition does not need to specify input and output features because they can be achieved from a GPU data structure (*FeaturesList*) automatically allocated and filled at allocation time. Figure 3 shows an implementation example, referring to *steerForSeparation* provided by the *OpenSteerWrapper* plug-in.

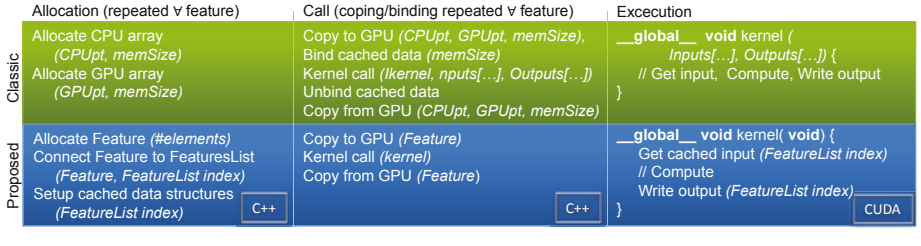

Fig. 2. Simplified representation of a functionality implementation – proposed approach compared to a classical approach. Items between brackets represent arguments.

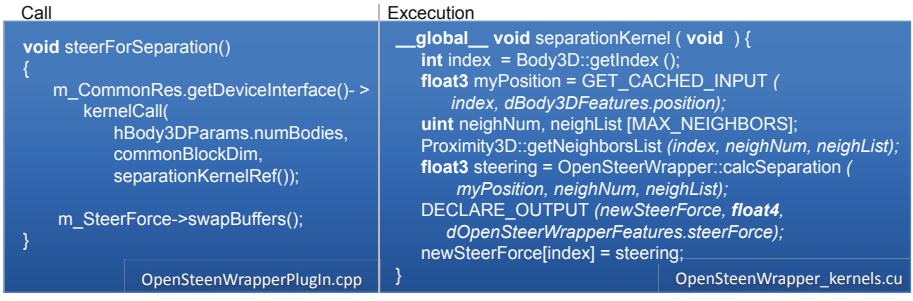

Fig. 3. Implemetation of *steerForSeparation* functionality. Here are the last two phases shown in Fig. 2. On the right, the execution is simplified thanks to features and functionalities furnished by previous plugins, e.g., getNeighborsList.

3.2 Usage

The following code shows how to create the behavior engine shown in Fig.1. The set of plugins is specified by means of a mixin classes technique [18]. The `BehaviorEngine` class contains the attributes and methods of each plug-in class.

```
// Allocation phase
OpenSteerWrapperPlugIn < Proximity3DPlugIn < EnvGrid3DPlugIn <
        Body3DPlugIn >>>> behaviorEngine;
// Update - Call and execution phases
behaviorEngine.reorderData();          // EnvGrid3D functionality
behaviorEngine.computeNeighborhood(); // Proximity3D functionality
behaviorEngine.steerForSeparation();  // OpenSteerWrapper functionality
behaviorEngine.applySteeringForce( elapsedTime );
// Draw (using the current OpenGL context)
behaviorEngine.draw();  // Drawable3D functionality
```

4 Core Plug-Ins

This section describes the core plug-ins of BehaveRT. For each plug-in, we show its features and functionalities. These plug-ins will then be used to demonstrate a mixin plug-in to present some case studies of runnable behavior engines.

Body3D Plug-In. This plug-in provides a basis for mobile individuals in virtual environments. It offers only one feature, representing the 3D position of individuals. The plug-in manages a common radius that approximates the space occupied by each individual in the world, with a sphere. A more sophisticated version of this plug-in could define more features, such as a different shape for the space occupancy approximation, e.g., a parallelepiped or a cube. On the other hand, a simpler version of this plug-in could be defined, such as the 2D version Body2D.

EnvGrid3D Plug-In. The EnvGrid3D plug-in provides one feature and some functionalities for environment subdivision and data reorganization. Environment subdivision offers a static grid subdivision of the world in cubic cells of the same size and enables swift identification of all individuals within a given radius. Data reorganization enables the aggregation of individuals belonging to the same cell in continuous regions of the GPU memory. The data reorganization fulfills two roles. First, it simplifies the neighbors search provided by the Proximity3D plug-in. Second, the operation improves the performance of the memory bandwidth during execution of the functionalities of other plug-ins. Because the interaction of individuals necessitates access to data on neighboring individuals, features content referring to the same individual is read many times. We therefore used a cached region of the memory of the GPU, i.e., textures, to improve the performance of frequently repeated data reads.

Proximity3D Plug-In. The Proximity3D plug-in provides a neighbor search operation for each individual in the community. This operation is fundamental because each individual must take decisions only according to its neighbors, and so it must be able to pick out efficiently these individuals. In order to obtain interactive results, this phase requires a careful choice of graphics hardware resources to obtain superior performance results and avoid the bottlenecks of a massive simulation. The output of this process is a new feature containing neighbors lists. Neighbors lists are encoded into a GPU-suitable data type that allows fast neighborhood queries. Each individual looks for neighbors in the cell where it currently is and in adjacent cells. Because the plug-in EnvGrid3D enables the memory data reorganization of features content, the cell content look-ups do not need additional data structures. Groups of agents belonging to the same cell will be located in continuous regions of the GPU memory. A simple linear search starting from a proper index based on the cell hash function is then sufficient. The plug-in also offers an internal functionality that allows other plug-ins access to individuals' neighbor lists. Further implementation information, including performance and scalability evaluations, on this approach are discussed in [4,5].

OpenSteerWrapper Plug-In. This plug-in provides a set of steering behaviors similar to those described by Reynolds in [12] and implemented in OpenSteer [13]. Steering behaviors implemented in the plug-in are applied to the whole community of individuals, while those in Reynolds's OpenSteer are applied independently to each individual. In our implementation, behavior differentiation among individuals can be achieved only by means of data-driven conditions. According to Reynolds's implementation, each steering behavior yields a steering force, represented by a 3D vector. The sum of steering forces of many steering behaviors is applied to the 3D movement of each individual.

Features provided by the OpenSteerWrapper plug-in include Forward, SteerForce, and Acceleration. The plug-in offers a set of functionalities representing steering behaviors, including steerForSeparation, steerForCohesion,

steerForAlignment, and steerForAvoidObstacle. It also provides functionality that applies steering force to individuals' positions (thus, modifying the Position feature provided by Body3D).

Drawable Plug-In. This plug-in provides by using a configurable rendering system a graphical representation of simulated individuals. Individuals can be rendered as simple points, billboards, or 3D models (by means of OpenGL geometry instancing). Positions and orientations are linked to VBOs and computed by ad-hoc OpenGL vertex and shader programs that directly generate a 3D representation of individuals.

5 Case Studies

This section introduces some case studies that applied to a large community of individuals generate impressive visual effects. Also we show how a community of individuals can interact with a generic environment. An ad-hoc BehaveRT plug-in, Shapes3D, provides picture and shape flocking by adding new features and functionalities on top of those described in the previous section. In both picture and shape flocking, individuals acts like a flock of bird by means of flocking behavior provided by the OpenSteerWrapper plug-in. In addition, each individual is associated with a target position and a target color. Per-individual target seeking is supplied by a functionality of the Shapes3D plug-in, while individual color managing is provided by the Drawable plug-in. The difference between picture and shape flocking is in the placement of target positions and the selection of target colors.

5.1 Picture Flocking

A community of individuals acting out picture flocking behavior recreates a 2D image in 3D space (Fig. 4a). Target positions are placed on a plane, and target colors are depicted from a picture in such a way as to associate each individual with a pixel of the picture. The Shape3D plug-in reads data from an input image file and initializes the target colors of individuals. When the simulation starts, individuals move toward the respective target positions, starting from random positions and moving in small groups by flocking. The Drawable3D plug-in also provides a smoothed translation of individuals' colors. For this reason, an individual attains the target color value only after several frames. When the simulation reaches a steady state, the result is a wavy image in space. This is due to the interaction between individuals that can never reach the exact target position because of inertia and separation behavior. As result, they simply wander near the target positions, aligned with and separated from their neighbors.

5.2 Shape Flocking

In shape flocking, behavioral target positions are placed on a 3D model surface. The Shapes3D plug-in reads a 3D model file and initializes the target positions to

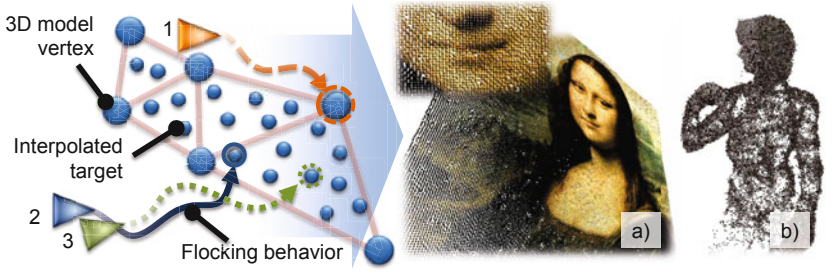

Fig. 4. A portion of the input 3D model and the path followed by three individuals during picture flocking (a) and shape flocking (b). The Shapes3D plug-in chooses target positions by randomly interpolating the positions of vertexes of an input 3D model. In this case, the target of individual 1 is a vertex, while targets of individuals 2 and 3 are interpolated points. Every individual has its own target position and moves toward that target exhibiting flocking behavior.

vertexes of the input 3D model. If the number of individuals is greater than the number of vertexes of the 3D model, Shapes3D interpolates the positions between vertexes. This generates a set of random points on the surface of each polygon of the 3D model (Fig. 4b). In the same way as picture flocking, each individual can be associated with a target color, depicted by the 3D model texture data. This characteristic is not currently implemented, but may be added to the Shapes3D plug-in initialization phase.

5.3 Shape Path Following

The 3D model can represent a path for the movement of individuals. Shape path-following behavior is an evolution of shape flocking. Only one characteristic differentiates these two behaviors: In shape path following, the target positions are not fixed. Indeed, target positions shift at every simulation step and effectively slide on the surface of the 3D model. The direction of the movement depends on the vertex order of the input model. Thus, given a set of ordered vertexes $v_1, v_2, ..., v_n$, the target position of agent j is the vertex v_i with $i = (j + \lfloor f * s \rfloor) \bmod |G|$ where f is the frame counter, s is the global speed of the path begin followed, and G is the set of simulated individuals (Fig. 5).

5.4 Environment Interaction

Many applications using autonomous characters require a method that allows individuals to interact with their environment. Here we present a new simple technique for the interaction of a large number of individuals with irregular terrain and static objects. The resulting effect (Fig. 7b) and performances are quite similar to those described in [16], with the difference that our implementation does not use a global navigation system and it is not limited to a 2D ground level. Thus, is possible to populate multi-floor environments, such as buildings.

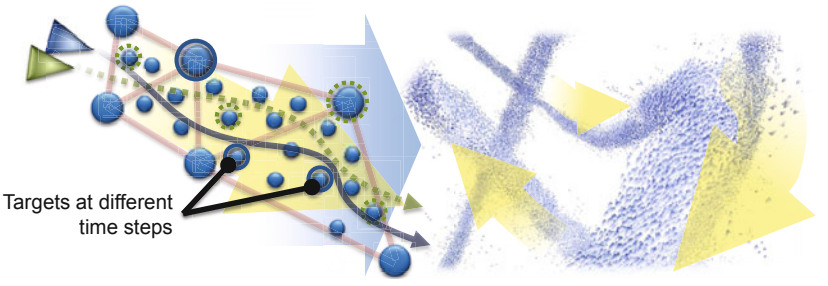

Fig. 5. The result of a community exhibiting shape path-following behavior, i.e., a flock of virtual birds moving on the surface of the input 3D model. At each step, the target positions change. They can be viewed as key points of a set of individual paths. The figure shows the paths of two individuals (continued and dashed) that wander around on the mesh of the 3D model.

An ad-hoc plug-in, Building3D, provides a feature that assigns a value to each cell of the environment grid (for simplicity we use the grid subdivision provided by the EnvGrid3D plug-in). A pre-computing phase samples the scene and checks for cells occupied by ground and other objects and assigns a binary value (empty/full) to a cell (Fig. 6a). This phase is similar to that described in [10] and [9] used for achieve navigation graphs from an approximate cell-decomposition of the navigable space. The purpose of the pre-computing phase is to achieve a discretized and simplified version of the scene. Individuals have access only to the information stored in their cell. The simulation phase is quick, because each individual has access to a restricted region of the memory shared among all individuals. By using modern commodity graphics hardware, the proposed technique permits the interaction of tens of thousands of individuals with a scene of medium complexity at interactive frame rates.

Additional Behaviors. Buildings3D provides the *floating behavior*, which allows individuals to float on irregular ground and objects. At each simulation step, an individual checks the value stored in its current cell and the value of its the future cell (Fig. 6b). Current and future cell information is used to modify the acceleration and the forward vector of characters (Acceleration and Forward are features provided by the OpenSteerWrapper plug-in). When an individual is not on the ground and the current cell is empty, then a downward factor is applied to the acceleration in order to simulate the gravitational force. When the future cell is full, impact with the ground is imminent. In this case an upward factor is applied to the acceleration (Fig. 6b) and the speed is reduced. However, there is a consequence to the speed reduction that occurs when individuals move up hills and mountains, in that individuals that reduce their speed become obstacles for approaching individuals. Through their separation behavior, approaching individuals steer to avoid these slowed individuals and move laterally. Owing to their alignment behavior, other individuals follow the group and avoid hills (Fig. 6c).

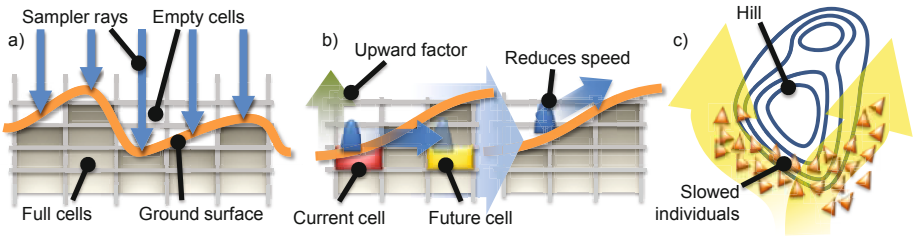

Fig. 6. (a) The pre-computing phase collects data on scene conformation. This scene is sampled with vertical rays. When a ray intersects the ground or an object, information are stored in the appropriate environment cell. (b) Floating behavior schema. Individuals check the current cell and future cell membership. In this example, the future cell data indicates the presence of the ground, and an upward force is applied to the character's acceleration. (c) An effect of the speed reduction in floating behavior.

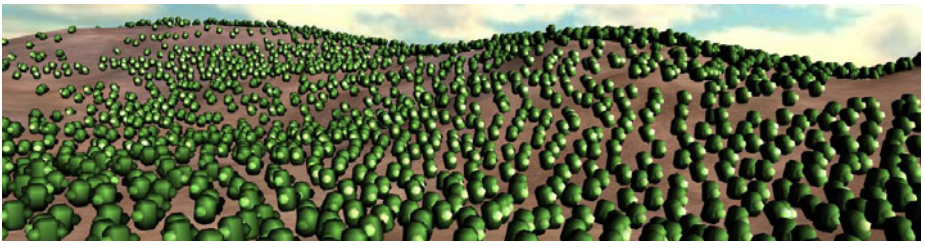

Fig. 7. Environment interaction screenshots. The scene has been created with Ogre3D graphics engine [7]. Individuals are rendered in a native OpenGL phase added to the Ogre3D's rendering queue.

In addition to the floating behavior, individuals interact by executing Reynolds's cohesion, alignment, and a modified version of separation behavior. Unlike the original version, the modified version of separation does not allow sudden changes in the steering force. It merely reduces the speed of individuals when the (actual) separation force is opposite to the forward direction (cross product close to -1). In contrast, modified separation behavior increases their movement speed when the original separation force and forward direction are similar (cross product close to 1).

6 Conclusions and Future Work

In this paper, we have introduced BehaveRT, a library that enables the definition of collective behaviors for large communities of individuals. The library provides an extensible and flexible structure that subdivides the definition of behaviors and data structures among well-separated and reusable plug-ins.

We designed a set of core plug-ins implementing the basis of a simulation system of individual-based modeling. In order to show the extensibility of BehaveRT, we introduced picture and shape flocking, two high-level behaviors for generating particular visual effects by coordinating the positions and colors of individuals. We also saw how individuals can interact with a generic virtual scene by means of an environment discretization phase and the execution of floating behavior.

BehaveRT also exploits the power of commodity GPUs by using well-known reorganization memory techniques and simplifying the usage of high-speed memory regions in this type of hardware. In fact, on commodity graphics hardware (Nvidia 8800GTS 512 Mb RAM, CUDA v3.0), picture and shape flocking run at 15 FPS with 130 K individuals (billboard-based rendering). Environment interaction case study runs at 15 FPS with 100 K individuals with billboard-based rendering and 30 K individuals with geometry instancing (about 80 polygons per individual). These results, similar to those provided by cutting-edge specific GPU-based simulations systems [14][16], suggest that BehaveRT exploits the computational power of the GPUs enabling developers/modelers to assign local behavioral models more complex for massive simulation and rendering. As consequence, we believe that the GPU acceleration provided in BehaveRT is applicable to video games because, like graphics and physics processing on the GPU, individuals based simulation is driven by thousand of parallel computation.

In the short term BehaveRT will be released as open-source but future work will equip BehaveRT with several new features: (i) Multiple communities. Currently, more than one community can be initiated and simulated at the same time, but individuals of a community cannot interact with individuals of another community. Thus, how an inter-community interaction system can be created remains unanswered. (ii) Environment interaction. The current system version of the individuals' interaction with generic scenery is based on an environment discretization scheme that associates a binary value (empty/full) to each cell. An extension could be represented by a system similar to JPEG image compression. Rather than use binary values, a set of known patterns could be used. Each pattern then approximates the portion of the scene contained in the cell.

References

1. Azahar, M.A., Sunar, M.S., Daman, D., Bade, A.: Survey on real-time crowds simulation. In: Pan, Z., Zhang, X., El Rhalibi, A., Woo, W., Li, Y. (eds.) EDUTAINMENT 2008. LNCS, vol. 5093, pp. 573–580. Springer, Heidelberg (2008)
2. Bleiweiss, A.: Scalable multi agent simulation on the gpu. In: GPU Technology Conference (2009)
3. Courty, N., Musse, S.R.: Simulation of large crowds in emergency situations including gaseous phenomena. In: Proceedings of the Computer Graphics International 2005, CGI 2005, Washington, DC, USA, pp. 206–212. IEEE Computer Society, Los Alamitos (2005)
4. Erra, U., Frola, B., Scarano, V.: A GPU-based method for massive simulation of distributed behavioral models with CUDA. In: Proceedings of the 22nd Annual Conference on Computer Animation and Social Agents (CASA 2009), Amsterdam, the Netherlands (2009)

5. Erra, U., Frola, B., Scarano, V., Couzin, I.: An efficient GPU implementation for large scale individual-based simulation of collective behavior. In: International Workshop on High Performance Computational Systems Biology, pp. 51–58 (2009)
6. Havok. Havok physics, http://www.havok.com/
7. Junker, G.: Pro OGRE 3D Programming (Pro). Apress, Berkely (2006)
8. Loscos, C., Marchal, D., Meyer, A.: Intuitive crowd behaviour in dense urban environments using local laws. In: Proceedings of the Theory and Practice of Computer Graphics 2003, TPCG 2003, Washington, DC, USA, p. 122. IEEE Computer Society, Los Alamitos (2003)
9. Maim, J., Yersin, B., Pettre, J., Thalmann, D.: Yaq: An architecture for real-time navigation and rendering of varied crowds. IEEE Computer Graphics and Applications 29(4), 44–53 (2009)
10. Pettre, J., Laumond, J.-P., Thalmann, D.: A navigation graph for real-time crowd animation on multilayered and uneven. In: First Int'l Workshop Crowd Simulation (2005)
11. PhysX. PhysX physics, http://www.nvidia.com/object/physx_new.html
12. Reynolds, C.: Steering behaviors for autonomous characters (1999)
13. Reynolds, C.: Opensteer - steering behaviors for autonomous characters (2004), http://opensteer.sourceforge.net/
14. Richmond, P., Coakley, S., Romano, D.M.: A high performance agent based modelling framework on graphics card hardware with cuda. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, Richland, SC, International Foundation for Autonomous Agents and Multiagent Systems, AAMAS 2009, pp. 1125–1126 (2009)
15. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.-m.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2008, pp. 73–82. ACM, New York (2008)
16. Shopf, J., Oat, C., Barczak, J.: GPU Crowd Simulation. In: ACM SIGGRAPH Conf. Exhib. Asia (2008)
17. Singh, S., Kapadia, M., Faloutsos, P., Reinman, G.: An open framework for developing, evaluating, and sharing steering algorithms. In: Egges, A. (ed.) MIG 2009. LNCS, vol. 5884, pp. 158–169. Springer, Heidelberg (2009)
18. Smaragdakis, Y., Batory, D.S.: Mixin-based programming in c++. In: Butler, G., Jarzabek, S. (eds.) GCSE 2000. LNCS, vol. 2177, pp. 163–177. Springer, Heidelberg (2001)
19. Sung, M., Gleicher, M., Chenney, S.: Scalable behaviors for crowd simulation. Comput. Graph. Forum 23(3), 519–528 (2004)
20. Taaffe, K., Johnson, M., Steinmann, D.: Improving hospital evacuation planning using simulation. In: Proceedings of the 38th Conference on Winter Simulation, WSC 2006, pp. 509–515 (2006)
21. Thalmann, D., Musse, S.R.: Crowd Simulation. Springer-Verlag New York, Inc., Secaucus (2007)