Computers & Graphics 37 (2013) 862–872

Contents lists available at SciVerse ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

A GPU-assisted hybrid model for real-time crowd simulations ☆

Murat Haciomeroglu [a,*], Oner Barut [b], Cumhur Y. Ozcan [b], Hayri Sever [b]

[a] Gazi University, Turkey
[b] Hacettepe University, Turkey

ARTICLE INFO

Article history:
Received 1 November 2012
Received in revised form
23 May 2013
Accepted 28 May 2013
Available online 19 June 2013

Keywords:
Real-time crowd simulation

ABSTRACT

In this paper, we propose two new techniques for real-time crowd simulations; the first one is the clustering of agents on the GPU and the second one is incorporating the global cluster information into the existing microscopic navigation technique. The proposed model combines the agent-based models with macroscopic information (agent clusters) into a single framework. The global cluster information is determined on the GPU, and based on the agents' positions and velocities. Then, this information is used as input for the existing agent-based models (velocity obstacles, rule-based steering and social forces). The proposed hybrid model not only considers the nearby agents but also the distant agent configurations. Our test scenarios indicate that, in very dense circumstances, agents that use the proposed hybrid model navigate the environment with actual speeds closer to their intended speeds (less stuck) than the agents that are using only the agent-based models.

1. Introduction

In recent years, the problems and possible solutions of real-time crowd simulation have attracted many researchers from a wide variety of research disciplines. One of the most fundamental problems of this area is the efficient autonomous navigation of thousands of virtual agents. The navigation of virtual agents in the simulation should be smooth (oscillation-free) and collision-free to imitate their real-world counterparts. In addition to the realism of the simulation, the techniques proposed should be as efficient as possible for real-time applications. The reason is that most real-time crowd simulations run on consumer level hardware. Despite the high process capabilities of modern consumer level computing devices, simulating the behavior of thousands of virtual agents in real-time is still a bottleneck for most of real-time simulations. Therefore, techniques proposed to increase the realism of a real-time simulation should be as efficient as possible, or use an additional resource in conjunction with the CPU. In this study, we take the latter approach. The proposed system clusters the agents by using the GPU's parallel processing capabilities, and then makes use of the cluster information to help agents determine better (lock-up free) paths towards their goal position. The test scenarios described in Section 5.1 indicate that when an agent uses cluster information, it reaches its destination quicker, with an average speed closer to its desired speed along its journey.

The clustering of virtual agents, which is the macroscopic information to be used by agents to query other distant agent configurations, is based on their positions and velocities. By using global cluster information, agents make their decisions based not only on nearby agents but also on distant agent configurations. For example, an agent might prefer to stay away from an incoming group (cluster) early on. To evaluate the proposed system, we present two test scenarios and compare the results with a state-of-the-art agent-based locomotion technique [1] (see Section 5.1).

The rest of the paper is organized as follows: Section 2 reviews the literature. Section 3 describes our proposed shader-based (GPU) agent clustering technique. Section 4 explains how the cluster information is used during real-time simulation. Section 5 describes the test scenarios and presents the experiment and performance results of the simulations. In Section 6, we conclude our study and discuss future directions.

2. Literature review

Crowd simulation brings many different components together, such as rendering, animation, path planning and navigation. Excellent surveys on each of these subjects, such as [2–5], and comprehensive applications, such as [6–8], abound in the literature. One can greatly benefit from understanding the different components and their relationships from these studies. The focus of this study, however, is only the navigation of virtual entities, which is one of the most fundamental parts of many crowd

☆This article was recommended for publication by M. Wimmer.

* Corresponding author. Tel.: +90 312 582 31 20.
E-mail addresses: murath@gazi.edu.tr, murathc@gmail.com (M. Haciomeroglu), onerbarut@cs.hacettepe.edu.tr (O. Barut), cumhuryigitozcan@cs.hacettepe.edu.tr (C.Y. Ozcan), sever@hacettepe.edu.tr (H. Sever).

0097-8493/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cag.2013.05.006

simulations. Therefore, we will narrow our discussion of crowd simulation to navigation issues.

Navigation is the ability of virtual agents to travel from one point to another smoothly and without collision in a given virtual environment. Two separate approaches to navigation have emerged in the past decade. The first one is the agent-based (microscopic) navigation model. In the agent-based model, for each agent, the navigation algorithm analyzes its local area and determines a collision-free trajectory at each step of the simulation. The Reynolds' rule-based microscopic navigation model [9] is an excellent example of agent-based navigation. Also, Helbing [10] defines a navigation model based on the social forces generated around each agent. Later on, van den Berg et al. [1] proposed a geometric solution for agent-based navigation which is smooth, efficient and conservative in terms of collisions. Each of the previously mentioned agent-based navigation models usually generates excellent trajectories for individuals. However, individual agents are ignorant of distant features such as an incoming dense group. Therefore, although individuals' trajectories are smooth in the local sense, their long-term paths may not be as smooth as they should be. In contrast, in a macroscopic navigation model (the second popular approach), such as the continuum-based model [11], individuals are aware of their distant surroundings. Thus, in the long term, smoother paths are generated. However, the main focus of the macroscopic techniques is the motion of the crowd as a whole, not the individuals motions. Therefore, the local trajectories of individuals are not as realistic as agents in a microscopic simulation. Although Narain et al. [12] attempted to bring individual trajectory quality to very dense scenarios by constraining the compression of virtual agents, local trajectories of individuals are still not as realistic as an individual-based simulation. On the other hand, individuals in an agent-based simulation have very limited information about other moving entities since they only take very close agents into account. This may lead the individuals to prefer unrealistic paths such as going towards crowded areas even though clearer paths exist. As a solution, considering the distant agents to calculate a better local path would greatly increases the computational overhead of the simulation.

The proposed model periodically clusters the agents on the GPU (in parallel) and serves this information to the agent locomotion algorithm that runs on the CPU. The cluster information helps the individuals in the simulation be aware of their distant surroundings without adding significant cost to the CPU. Similar to the proposed approach, Guy et al. [13] clustered the distant agents based on their positions, then successfully incorporated this information into an agent-based navigation system. However, their technique requires the clusters to be determined for each individual separately on the CPU, and this increases the cost of the simulation. Similarly, Golas et al. [14] cluster the distant agents and insert the clusters in the agents' velocity space as a constraint. However, their method requires agents indexed on a kd-tree and their clusters are not grouped based on agent velocities but only based on positions, therefore, determined macro-information (clusters) is not always accurate. Recently, Haciomeroglu et al. [15] determined cluster information on the GPU and used this information for relaxing the simulation from densely packed areas. However, unlike the proposed technique, their method neither calculates the cluster centers nor the average velocity of the clusters. Therefore, their agents have very limited macroscopic information and can only be used for local decisions. Nevertheless, the aforementioned studies show that the cluster information can greatly help the simulation in terms of finding better semi-global paths for the agents.

Previously mentioned navigation methods are either designed for CPU or CPU+GPU (very little support from the GPU). Some researchers have adopted navigation algorithms to the parallel GPU architecture [16–18]. Pure GPU-based models can simulate the navigation behavior of very large crowds; however, little computation power is left for rendering the real-time simulation environment on the GPU. Therefore, our approach divides the cost of the simulation between the CPU (agent-based model) and the GPU (clustering).

Recently, GPU-based data clustering has become an attractive area for study. There are several studies that use GPU for clustering purposes. Hall and Hart [19] performed traditional k-means iterations on the GPU. Their technique uses textures as a data transfer tool for the input data. They calculate the clustering results, such as distances to cluster centroids, via fragment shaders in multiple passes. Cao et al. [20] calculated distances to all cluster centers in one pass and labeled the input data with each cluster's unique identification number using multiple-pass rendering. Shalom et al. [21] used the texture swapping technique to reduce the data transfer overhead between the CPU and the GPU that occurs among clustering iterations. Neither the previously mentioned GPU-based methods nor the common clustering algorithms are suitable for our purposes because of two concerns. First, classical clustering approaches require cluster count prior to the clustering process; second, their goal is to place all the data into one of the clusters. However, in our method, the number of clusters that will be formed after a clustering process is unknown prior to the process because our motivation is to generate clusters from agents that are similar to each other by means of a distance metric (see Section 3.2), which causes the clustering process to form a different number of clusters each time.

3. GPU clustering

To perform the agent clustering operations on the GPU, we employ a method inspired by Haciomeroglu et al.'s [15] work. In [15], textures are used to store the agents information in order to calculate clusters on the GPU. Because of the fact that their texture has double the height and width of their uniform grid which covers the simulation environment, each of their grid cells has four texels to store the information of the agents within the cell. Considering each agent requires two texels to store its information, every grid cell can contain two agents at the same time. Hence, they employ multiple identical textures each overlapping with the others and also the grid respectively to address more than two agents in the same grid cell. Therefore, in [15], textures may contain lots of irrelevant data when many of the grid cells are empty. The proposed method prevents this unnecessary data transfer overhead.

We also expand [15] to calculate not only the clusters, but also some additional information regarding the clusters, such as the cluster position, which is the center of mass of the cluster, and average cluster velocity, which is an interpolation of velocities of the cluster members. In addition, the proposed method performs the cluster construction on the GPU. For a detailed performance analysis, see Section 5.2.1.

The proposed clustering method is strongly related to the agglomerative form of the hierarchical clustering [22]. Agglomerative hierarchical clustering methods assume that every single data point is a cluster on its own at the beginning, and then they iteratively combine the two most correlated clusters at a higher level of the hierarchy tree until there is only one cluster as the root. Then, the tree is cut at the level that yields the desired number of clusters. Our clustering method differs from this approach in two points. First, we do not have a pre-defined number of clusters. Therefore, rather than completing the entire tree, we stop advancing towards the root when there are no more nodes to combine that satisfy our correlation function. Second, even though it is originally an iterative algorithm, we implement it in parallel on the

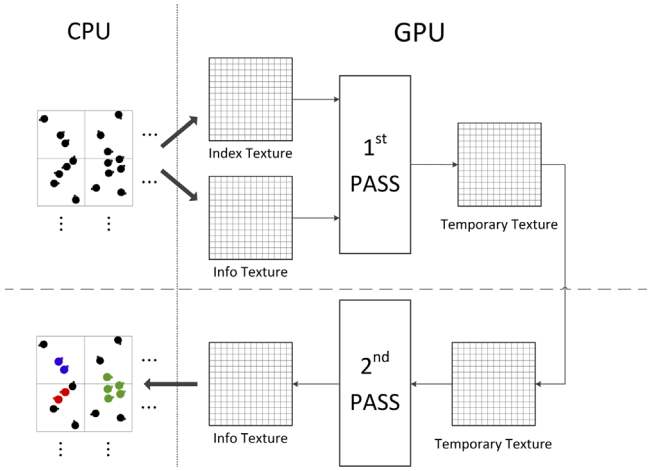

Fig. 1. System overview of the proposed technique that uses two consecutive GPU passes.

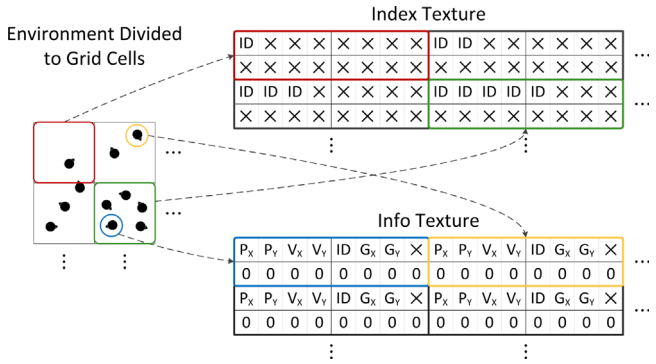

Fig. 2. General overview of the texture initialization stage of the proposed technique.

GPU. Our algorithm runs for every agent (in parallel) to find other agents that can be grouped in a cluster.

The presented technique is implemented using OpenGL and GLSL to perform the clustering operations. Although the presented technique can be implemented by using CUDA or OpenCL with a possible performance hit, those technologies are not available in every platform.

In the proposed technique three RGBA textures with 32-bit floating points per channel are employed as data transfer utilities between the CPU and the GPU. Clustering results are calculated in the geometry and fragment shader stages of the graphics pipeline. To determine clusters, cluster centers and average cluster velocities, our method performs two consecutive GPU pipeline passes in each clustering process. Clustering results are written to a frame buffer object (FBO), which can be considered as an output buffer on the GPU memory and read back to the CPU after every clustering process. A general overview of the system is illustrated in Fig. 1.

3.1. Clustering textures

To perform the clustering operations (and for indexing purposes for use by the locomotion algorithm), we setup a grid structure that divides the simulation environment into uniformly distributed square grid cells. It should be noted that, because of the uniform grid partitioning, many empty grid cells might be sent to the GPU if the distribution of the agents is non-uniform.

The first texture – which will be referred as the index texture from now on – actually contains the index (grid) information. We use four texels for each grid cell, and we position these texels such that the first two will be placed on top of the next two, as illustrated in Fig. 2. Agent IDs in every cell are written to one of the four texels that belong to that cell. The index texture is updated by individual agents on the CPU while they navigate the environment.

The second texture we use is the info texture, which contains the position, velocity, agent ID and grid position information of each agent. The grid position is necessary to quickly access other agents' IDs from the index texture. By using agent IDs, agent information is easily accessible as the agents are ordered by their IDs in the info texture. In the info texture, there are four texels reserved for each agent, as illustrated in Fig. 2. Two of these texels are used to store agent information, while the other two are empty and will be used for storing the results of the clustering process (See Section 3.2). The info texture is updated by the locomotion engine (on the CPU) once before each clustering process, as the agents' information (position and velocity) changes.

The third texture, with the same dimensions and format as the info texture, is actually used as a temporary data storage between the two GPU passes of the clustering process (see Fig. 1), because the graphics pipeline does not allow one of the input textures to be written on via shaders. For this purpose, the temporary texture is bound to the FBO as the output buffer and the output of the first pass is written on the FBO. Then the info texture and the temporary texture are swapped before the second pass, making the temporary texture the input and the info texture the output of this pass. When the first pass is completed, the temporary texture will also contain one of the other agents' ID who satisfies the clustering metric. Among all agents that satisfy the metric, the one with the lowest ID is chosen (see Section 3.2).

It should be mentioned that one of the drawbacks of our method is the limitation of the number of agents that can be in one grid cell for clustering. We developed and tested our system with AMD HD6970 graphics card that supports 32 texture units with 32-bit RGBA color channels and dimensions up to $16K \times 16K$. When the index texture has maximum dimensions of $16K \times 16K$, we can use 2^{28} texels, which can index up to 2^{30} agents, because each texel can contain four agent IDs. However, our simulation environment uses a 64×64 grid and we have devoted four texels for each grid cell. Hence, the number of agents whose IDs can be indexed into the same grid cell area is limited to 16 in this study. If there are more than 16 agents in the same cell at the same time, our clustering algorithm ignores the agents who entered the cell area after the 16th agent. This fact may affect the quality of the clustering stage output if the grid cell size is large. Another limitation is the total number of agents that can be processed by the clustering algorithm. The info texture stores and transfers the agents' information to the GPU. When we use the info texture with its maximum dimensions of $16K \times 16K$, we can hold maximum 2^{26} agent information at one time, because each agent requires 4 texels to store its information.

3.2. Clustering passes

In our method, each cluster has a unique cluster id and each clustered agent holds its cluster id. The clustering algorithm is run in parallel and it is hard to define a unique id for each cluster in parallel. To overcome this problem, we define each cluster by the id of the agent in the cluster who has the lowest id. This agent is referred as the cluster leader.

Each agent in our clustering model is handled by a GPU processor in parallel. The main drawback of this approach is that, although they read from a shared memory, each processor should have a private area to write its output. As neither the number of the clusters to be formed nor the agent count in a cluster is known prior to the clustering process, it is not possible to reserve enough area for each processor to write down all of the data it acquires.

Therefore, each processor records only a unique identifier which represents the cluster the agent will belong to. These unique identifiers are called cluster ids and they actually correspond to the lowest agent id in that cluster.

Algorithm 1. The algorithm of the first clustering pass on the fragment shader.

```
Input: Info (info texture)
    Index (index texture)
    x, y (current pixel's coordinates)
Output: Temporary (temporary texture)
if y % 2==1 then
|return;
end
if Info[x,y].alpha! = null then
| Temporary[x, y] = Info[x, y];
| return;
end
temp_leader_id=Info[x + 1, y].red;
grid_pos.x=Info[x + 1, y].green;
grid_pos.y=Info[x + 1, y].blue;
A=Check the current cell (grid_pos) and neighboring cells and
retrieve agent id's that are found;
for all the agent id a_i in A do
| if a_i and current agent satisfy Metric function then
| | if a_i's id < temp_leader_id then
| | |temp_leader_id = a_i's id;
| | end
| end
end
Temporary[x, y]=Info[x, y];
Temporary[x + 1, y]=(Info[x, y].red, temp_leader_id, null, null);
```

As mentioned before, each clustering process consists of two rendering passes that run one after the other. In the first pass, a separate fragment shader (whose pseudo-code is given in Algorithm 1) instance runs for each agent to process other agents that are in the same or neighboring grid cells. Thus every agent can find the agents that are below a given threshold according to the correlation function (Eq. (1)). To run a fragment shader for each agent, the info texture is mapped onto a rectangle primitive that has the same height and width as the info texture. Then, by arranging a viewport size equal to the size of the info texture, and setting an orthographic projection, each texel of the info texture corresponds to a different pixel of the FBO. This ensures that each texel of the info texture will be processed by a separate fragment shader instance. It should also be noted that since agent information is stored in four texels, only the first of them triggers the clustering algorithm.

$$Metric = \delta\|\overrightarrow{P_l R_l}\| + \alpha\|\overrightarrow{P_v} - \overrightarrow{R_v}\| \quad (1)$$

Every agent (shader) tries to find other agents that are similar in terms of position, direction and speed. Similarity is determined by using Eq. (1), which is the correlation function. In Eq. (1), P_l and P_v are the position and velocity vector of the current agent, and R_l and R_v are the position and velocity vector of a neighboring agent respectively. Also in Eq. (1), δ and α are the coefficients of the metric function, and are set to 1.0 in this study. If the *Metric* variable is less than a threshold value, then the agent R is associated with the current agent, P (i.e., they should be in the same cluster). If the shader instance finds more than one agent to be clustered with, it discards the ones with higher IDs and picks the one with the lowest ID as its cluster leader. Although this process would cause multiple leaders in a single cluster (since an agent's leader may consider another agent as its leader) at the end of second pass, as discussed below, eventually the agent with the lowest ID will be regarded as cluster leader by all the members of the same cluster. Finally, the determined cluster leader IDs are written to the temporary texture, which is bound to the (current) FBO using fragment shader instances. However, not only the leader IDs but also the positions and velocities of the agents are written to the temporary texture, as illustrated in Fig. 3.

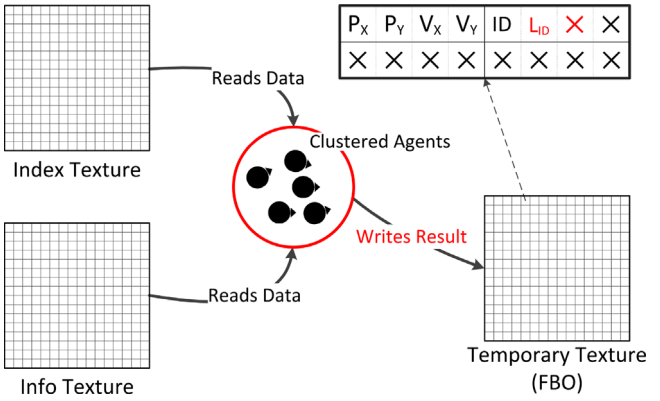

Fig. 3. General overview of the first GPU pass of the proposed technique. Each agent (fragment shader) copies its own data that is stored on the info texture and writes this copy to the temporary texture by changing only the second and third components of the second texel.

After the first pass, to eliminate the texture transfer overhead between CPU and GPU, we employ texture swapping, which sets the input texture of the previous rendering pass (info texture) as the output texture of the following rendering pass, and the output texture (temporary texture) of the previous rendering pass as the input texture of the following rendering pass. Thus, the info texture of the first pass is bound to the FBO and used as output buffer. However, we do not have to clear the output buffer before starting to the second pass because we use the empty texels under the texels that contain agent information on the info texture (see Fig. 2). Thereby, we also gain an increase in performance by not writing zero values to all the texels in the info texture.

Algorithm 2. Second clustering pass on geometry and fragment shaders.

```
Input: Temporary (temporary texture)
    x, y (texture coordinates of current agent)
Output: Info (info texture)
leader_id=Temporary[x + 1, y].green;
leader_pos=Get leader's position using leader_id;
while Temporary[leader_pos.x, leader_pos.y].green! = leader_id
do
| leader_id = Temporary[leader_pos.x, leader_pos.y].green;
| leader_pos = Get leader's position using leader_id;
end
vertex1.color=Temporary[x, y];
vertex1.pos=(leader_pos.x, leader_pos.y + 1);
emit(vertex1);
vertex2.color=(0, 0, null, 1);
vertex2.pos=(leader_pos.x + 1, leader_pos.y + 1);
emit(vertex2);
vertex3.color=(Temporary[x + 1, y].red, leader_id, null, 0);
vertex3.pos=(x + 1, y + 1);
emit(vertex3);
```

The second pass (whose algorithm is given in Algorithm 2) is run immediately after the first one, because there is no need to

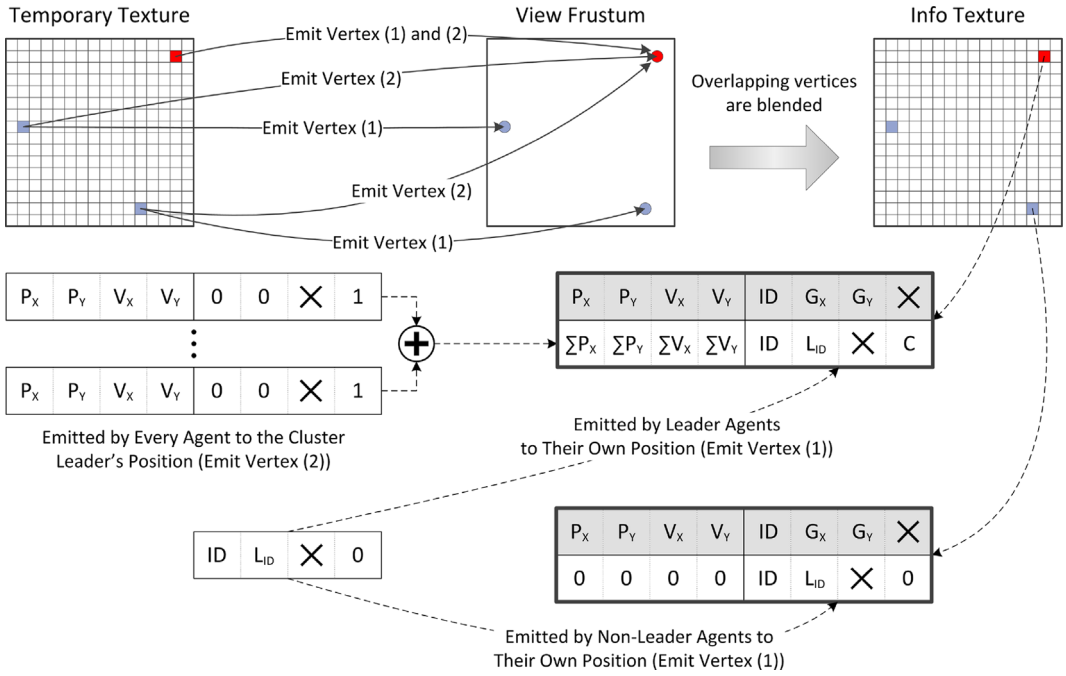

Fig. 4. General overview of the second GPU pass of the proposed technique. Red texels in the temporary texture and the info texture indicate the leader agents of the cluster while blue ones indicate non-leader agents. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

transfer data from the GPU back to the CPU. The first objective of the second pass is to solve the multiple leader problem mentioned previously by merging the distributed clustering information that is obtained in the first pass. Most of the work in this pass is done by the geometry shader instances. To trigger a separate geometry shader instance for each agent, a point primitive is created. Then, each geometry shader instance searches for the real leader ID for its current agent using the temporary texture. To determine the real leader of the current agents cluster, the current agent compares its leader ID with its leader's leader ID (only the real cluster leaders have the same leader ID as their own ID). If they are equal to each other, the current agent stops searching for the cluster leader. But if they are not equal, the current agent sets its leader ID as its leader's leader ID and then continues searching until it finds the real cluster leader.

It should be noted that this technique has a transitivity effect while clustering the agents (see [15] for details). This may – in rare occasions – cause a transitivity chain; a chain of agents where every agent pair has slightly different velocity and position values from each other. This effect may seem to be a false outcome because two distant agents in the transitivity chain may have completely different velocity and position values. But it actually is essential for the steering methods described in Section 4. The agents in a large cluster of transitivity chain are not affected by this phenomenon since the cluster information generated is used by the agents that are not in that cluster. The agents that are not in the cluster, however, will actually be using this situation to their advantage as they should be aware of the cluster and steer away from it. For example, the agent group may have formed a perfect circle, and they may have the exact same velocity or same speed with a direction of movement towards the center of the circle. Either way, the proposed system will put these agents into one big cluster therefore other agents will be able to recognize this big group. Breaking the chain at any point means putting two closely related agents into separate clusters which will mislead the agents that are not in that cluster.

After determining the real cluster leaders for the agents, all agents write their real leader IDs to the empty texels in the info texture currently bound to the FBO. Hence, all agents emit a vertex whose position overlaps with one of the empty texels. The front color value of this vertex holds the real leader ID information to carry to the FBO, as illustrated in Fig. 4.

The second set of objectives of the second pass is to find cluster centers and average cluster velocities. To achieve these objectives, position and velocity information of each cluster member must be summed within the clusters. We gather the sum of all cluster members' positions and velocity vectors and store them in the empty texels (along with the leader ID values) of the info texture. However, aggregated cluster information is stored only in the empty texels of the cluster leaders. For this purpose, leader agents emit two vertices whose front color values contain the position and velocity information of themselves; these vertices are positioned to overlap with their empty texels. Similarly, non-leader agents emit two vertices carrying their position and velocity information. However, non-leader agents emit their vertices not positioned to align with their empty texels but positioned to align with their real leader agent's empty texels (as illustrated in Fig. 4). Thus, all vertices emitted by the same cluster's members are overlapped at the empty texels of that cluster's leader agent. All those overlapped vertices' front color values are added without any overhead thanks to the hardware support of blending on the GPU. The blended color values contain aggregated position and velocity vector information for the cluster. While each cluster member emits their position and velocity vector, they also send the value 1 to the alpha value of the second empty texel. Thus, by using blending, the cluster member count for each cluster is also calculated. In the final FBO, the alpha values are summed (blended) to determine the number of members in each cluster. After this information is transferred back to the CPU, the aggregated values will be divided by the cluster member count. Thus, the center of mass and the mean velocity vectors are determined for each cluster.

4. Using the cluster information

We have discussed how the proposed clustering method efficiently determines the clusters using the graphics processor.

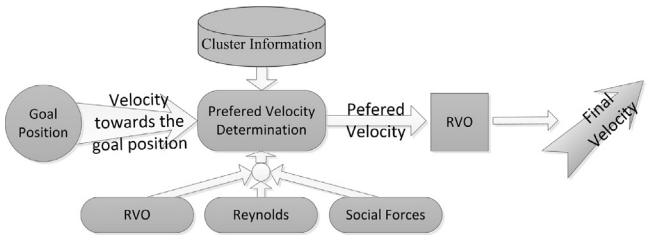

Fig. 5. Overview of the proposed agent locomotion system. Key: RVO is the Reciprocal Velocity Obstacles technique, Reynolds is the Reynolds Steering Behaviors technique and Social Forces is the Social Forces technique.

In this section, we explain how the cluster information is used in agent navigation. The idea is simple. If an agent determines that a cluster is about to cross its path, then the agent must find a new direction. If an agent is a member of a cluster, then the agent does not consider its own cluster as a potential obstacle in its path.

The proposed system consists of two stages (as illustrated in Fig. 5). In the first stage, a preferred velocity, $\overrightarrow{PV_i}$, is determined for agent i using nearby clusters. The preferred velocity is calculated using the following equation:

$$\overrightarrow{PV_i} = G\hat{D}_i + \hat{D}_i + \overrightarrow{CF_i} \tag{2}$$

In Eq. (2), $G\hat{D}_i$ is the unit force vector directed to the goal of the agent, $\hat{D}_i$ is the unit force vector indicating the current direction of the agent and $\overrightarrow{CF_i}$ is the cluster force vector calculated by using three different techniques which are explained in Sections 4.1–4.3. Then $\overrightarrow{PV_i}$ is normalized then scaled by the agents intended speed to determine the final $\overrightarrow{PV_i}$. Hence, it should be noted that the long term guidance vector ($\overrightarrow{PV_i}$) does not contains any change in speed information (only guidance direction information). The determined preferred velocity is then used as input to the Reciprocal Velocity Obstacles (RVO) [1] technique. The RVO technique requires a preferred velocity, and tries to find the closest collision-free velocity to the preferred velocity. As is seen, cluster force does not directly enforce the final velocity; it only guides the agents along their path to the goal position. In the following sections, the method used to determine the cluster force, $\overrightarrow{CF_i}$, is explained. We used three different methods for cluster force determination. The first one is based on the RVO, the second one is based on the Reynolds Steering Behaviors [9] and the third one is based on the Social Forces technique [10].

4.1. RVO

In the RVO based technique, a preferred velocity is input to the algorithm, which is determined by multiplying the normalized goal direction and the intended speed of the agent (such as 1.4 m/s). When calculating a collision-free path using the cluster RVO, only the clusters (as single fat agents) are used as the nearby agents. The diameter of an agent (cluster) is taken, in meters, as one-third of its member count. This is a heuristic estimate which assumes that crowded clusters will likely occupy more area than less crowded clusters. In the test scenarios (see Section 5.1) the time horizon parameter of the RVO is increased to 5 times the original value. The reason for that is an agent also can sense 5 times farther (50 m away) agents (clusters in this case) than the agent based RVO. Thus, the cluster RVO can take into account distant clusters. The resulting velocity vector obtained from the cluster RVO is then used as the cluster force, $\overrightarrow{CF_i}$, in Eq. (2).

4.2. Reynolds

In the Reynolds Steering Behaviors [9] based technique, every agent tries to find possible future collisions with other clusters (again, the diameter of the clusters are taken, in meters, as one-third of their member count), if the agent continues to move in its current direction. If more than one collision is found, the collision that is likely to occur first is chosen. Then the agent turns left or right based on its direction relative to the cluster's direction. The determined final velocity is used as the cluster force, $\overrightarrow{CF_i}$, in Eq. (2).

4.3. Social Forces

In the Social Forces based technique [10], every nearby cluster generates a push force on the agent. However, only the clusters that are not similar to the current agent generate a push force on the current agent. Similarity is determined by subtracting the cluster velocity, $\overrightarrow{V_c}$, from the current agent velocity, $\overrightarrow{V_a}$. If the magnitude of the resulting vector, $\|\overrightarrow{V_a} - \overrightarrow{V_c}\|$, is smaller than 0.5, we consider the cluster as similar to the current agent and do not allow the cluster to generate a push force. This is because we do not want the agents to stay away from the flow that is similar to their own velocity. For non-similar clusters (in terms of velocity) a push force, $\overrightarrow{CF_i}$, for the agent i is calculated by using the following equation:

$$\overrightarrow{CF_i} = \sum_{j=0}^{n} \left(\hat{F}_j \frac{|\hat{F}_j \cdot \hat{D}_i|}{D_{ij} \times \frac{1}{\sqrt{S_j}}} \right) \tag{3}$$

In Eq. (3), $\hat{F}_j$ is the normalized vector generated from the jth nearby cluster position to the agent position, $\hat{D}_i$ is the normalized direction of agent i, D_{ij} is the distance between agent i and the jth nearby cluster center and S_j is the number of agents that the jth nearby cluster contains. Eq. (3) defines the critical interactions between the agents and clusters, such as the following: Distant clusters generate less powerful push forces than the closer clusters; crowded clusters likely have larger radii, hence generate more powerful push forces; and the clusters just beside the agent i generate less powerful forces than the clusters located in front or behind. The reason for the last enforcement is to encourage the lane formation in the simulation. The final vector, $\overrightarrow{CF_i}$, is used in Eq. (2).

One possible scenario that illustrates the difference between the original RVO and the proposed system is given in Fig. 6. The agents with the cluster information make better navigational decisions than the agents without the cluster information because they take into account the distant agents thanks to the cluster information determined on the GPU.

5. Results and analysis

To evaluate the model presented in this paper, in terms of both trajectory quality and performance, we compare our clustering based hybrid navigation model with bare RVO (which is labeled as unclustered in the results given because of performing only short-range planning and having no clustering calculations). In Section 5.1, we discuss experimental results to prove that long-range planning by utilizing cluster information improves the navigation quality of the crowd compared to unclustered results. In Section 5.2.1, we assess the performance of our GPU based clustering method among the other possible clustering solutions. In Section 5.2.2, we evaluate the overhead of long-range planning by comparing our two stage locomotion model (long-term + short-term planning stages) with unclustered (only short-term) results.

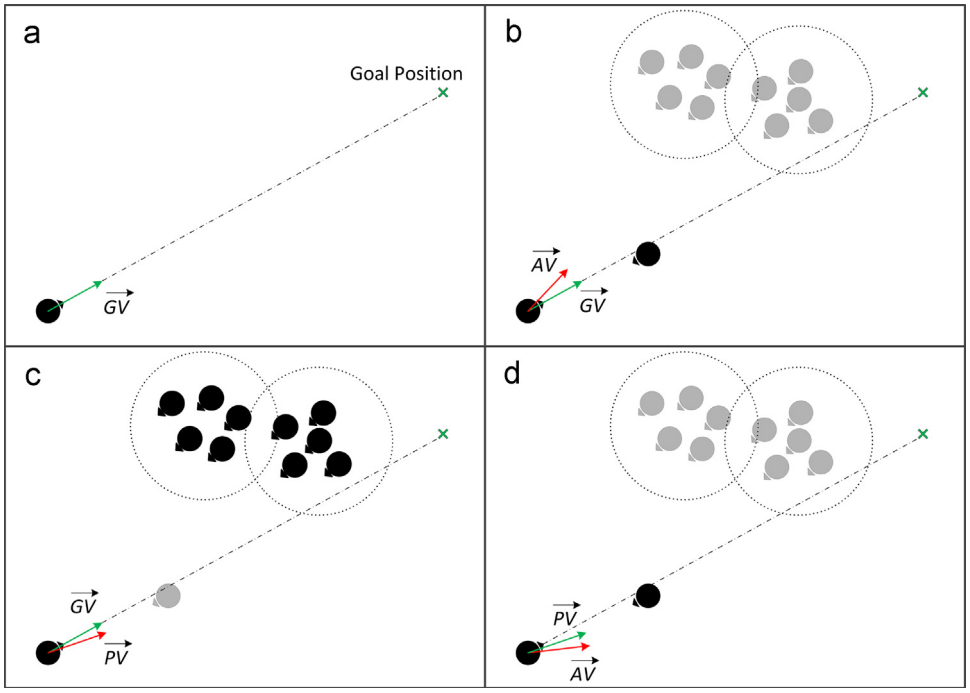

Fig. 6. Four figures describing the difference between the classical microscopic simulation (RVO) and the proposed model. (a) The subject agent calculates a preferred velocity which is directly determined by using the goal position ($\overrightarrow{GV}$). (b) RVO only locomotion scenario in which the subject agent does not know the distant agent configurations; hence, it calculates a non-optimized actual velocity ($\overrightarrow{AV}$) (towards the incoming groups). (c) The proposed system calculates the preferred velocity ($\overrightarrow{PV}$) taking into account the distant clusters; therefore, an optimal $\overrightarrow{PV}$ is calculated. (d) As the second stage in the proposed system, RVO takes the previously determined $\overrightarrow{PV}$ (by using distant clusters) and calculates a collision-free $\overrightarrow{AV}$.

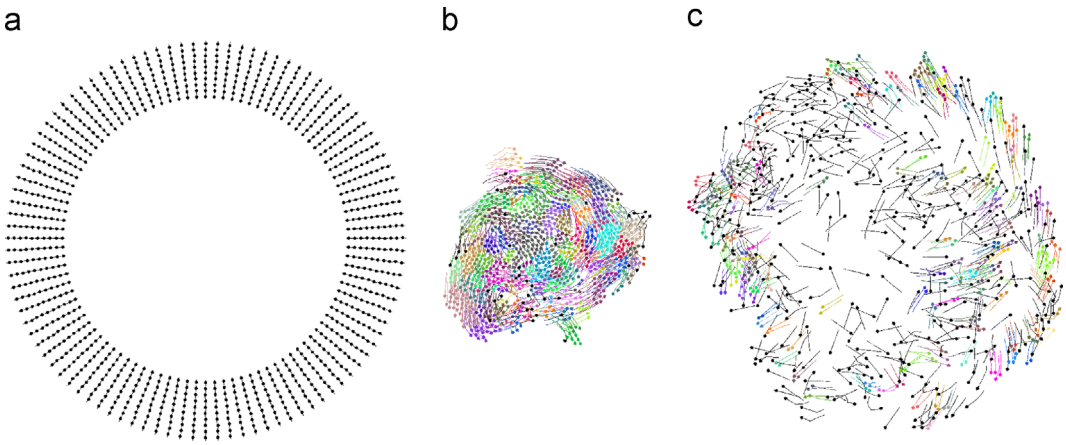

Fig. 7. An illustration of Test Scenario 1 in which the agents are initially positioned on a circle of 50 m diameter. Their destination position is set as the opposite side of the circle. Each color indicates a different cluster. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

Finally, in Section 5.2.3, we include the overall performance results (including rendering) to show the effect of our model on the FPS value of a real time system.

5.1. Experiment results

The proposed techniques and the original RVO (unclustered) technique are tested using two scenarios. In both scenarios, every agent is positioned at an initial position and a goal point is assigned to each of them. The main assumption is that more organized agents will, on average, arrive at their destinations quicker and navigate faster than the less organized agents. Therefore, the agents' average destination arrival time and average speed while navigating through the goal point are measured and the results are presented in Sections 5.1.1 and 5.1.2. Each scenario and each technique is also simulated with different numbers of virtual agents (100, 200,…,1000).

5.1.1. Test scenario 1

In the first scenario, every agent is positioned symmetrically in a circle, as shown in Fig. 7. Goal points for the agents are on the opposite side of the circle. In Table 1, the agents' average navigation time is shown, and in Table 2, the agents' average speed is presented for each technique. As shown in Tables 1 and 2, when using cluster information, agents reach their destination quicker and their average velocity is higher than the agents in the unclustered simulation. In this test scenario, each technique that uses cluster information produced similar results in terms of both destination arrival time and average velocity. The data in Tables 1 and 2 also showed that the benefit of the proposed system increases as the density of agents increases.

5.1.2. Test scenario 2

In the second scenario, each agent is positioned on one of the edges of a square, shown in Fig. 8. The goal points of the agents are at the opposite side of their current edge. Additionally, several obstacles are positioned in the center of the field. In Table 3, the agents' average navigation time, and in Table 4, the agents' average speed is presented for each technique. Similar to Scenario 1, in Scenario 2, when using cluster information, agents not only reach their destination quicker but their average velocity is also higher (see Tables 3 and 4). Among the clustered simulations, the Social Forces based technique produced the best results, on average, in terms of both destination arrival time and average speed. The data in Tables 3 and 4 also show that the benefit of the proposed systems increases as the density of agents increases.

5.2. Performance results

5.2.1. Comparison of GPU clustering methods

The clustering of virtual agents on the GPU is one of the crucial points in the presented system. As indicated before, our clustering method is an improved version of a previous study [15]. To clarify the differences, we will discuss requirements of both techniques considering both simulations have 1000 agents and 64×64 grid. In [15], 8 textures are required for clustering while the proposed method requires only 2 textures for input. A single texel in both techniques requires 16 bytes of memory. In total, [15] allocates $8 \times 128 \times 128 \times 16$ bytes of total memory for all 8 textures. On the other hand, our technique uses $128 \times 128 \times 16$ bytes for index

Table 1
Scenario 1—average navigation times (s) of the agents for each technique.

Number of agents	Unclustered	RVO	Reynolds	Social Forces
100	85.61	81.01	82.73	80.75
200	103.55	89.30	92.31	91.43
300	123.58	98.17	101.05	99.95
400	145.93	109.38	114.39	104.07
500	149.51	119.65	129.26	116.81
600	170.66	124.48	130.79	123.05
700	192.73	135.52	128.91	122.63
800	183.64	146.51	143.16	130.07
900	208.32	143.61	157.05	125.94
1000	202.91	152.91	156.56	138.42
Average	156.64	120.05	123.62	113.31

Table 2
Scenario 1—average speed (m/s) of the agents for each technique.

Number of agents	Unclustered	RVO	Reynolds	Social Forces
100	1.20	1.29	1.27	1.30
200	1.09	1.22	1.21	1.24
300	0.99	1.19	1.17	1.21
400	0.89	1.14	1.10	1.21
500	0.91	1.11	1.05	1.18
600	0.84	1.10	1.07	1.15
700	0.77	1.08	1.09	1.20
800	0.81	1.05	1.05	1.16
900	0.77	1.08	1.02	1.20
1000	0.78	1.06	1.03	1.16
Average	0.91	1.13	1.10	1.20

Table 3
Scenario 2—average navigation times (s) of the agents for each technique.

Number of agents	Unclustered	RVO	Reynolds	Social Forces
100	99.14	92.51	91.33	91.20
200	112.26	98.48	103.48	97.82
300	144.94	112.66	113.17	102.23
400	339.73	127.11	129.24	109.38
500	179.83	163.29	142.07	115.19
600	242.56	140.44	138.72	114.65
700	213.48	146.14	148.89	122.82
800	259.21	151.08	163.40	125.74
900	224.69	165.11	162.24	130.35
1000	322.60	176.24	174.47	136.84
Average	213.84	137.30	136.70	114.62

Table 4
Scenario 2—average speed (m/s) of the agents for each technique.

Number of agents	Unclustered	RVO	Reynolds	Social Forces
100	1.16	1.26	1.26	1.27
200	1.07	1.21	1.18	1.24
300	0.92	1.13	1.13	1.20
400	0.58	1.04	1.04	1.17
500	0.80	0.89	0.98	1.14
600	0.67	1.00	0.99	1.15
700	0.71	0.98	0.97	1.11
800	0.66	0.97	0.92	1.11
900	0.69	0.93	0.94	1.09
1000	0.58	0.91	0.90	1.07
Average	0.78	1.03	1.03	1.16

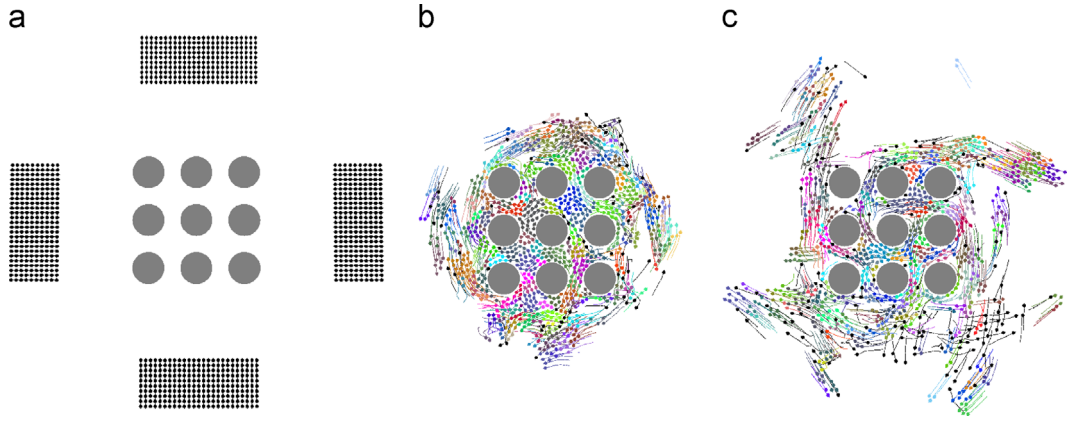

Fig. 8. An illustration of Test Scenario 2 in which the agents are initially positioned on each side of a square. Their destination is set on the opposite edge of the square. Each color indicates a different cluster. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

Table 5
Scenario 1—average time (ms) required for each clustering technique.

Number of agents	Haciomeroglu et al.'s method	CPU+GPU (single pass)	Pure GPU (two passes)
500	3.82	0.84	0.70
1000	4.18	1.23	0.80

Table 6
Scenario 2—average time (ms) required for each clustering technique.

Number of agents	Haciomeroglu et al.'s method	CPU+GPU (single pass)	Pure GPU (two passes)
500	3.71	0.79	0.73
1000	3.98	1.07	0.85

Table 7
Scenario 1—average performance overheads ratios (%) for each technique with respect to unclustered simulation.

Number of agents	RVO	Reynolds	Social Forces
100	41.69	39.41	32.82
200	55.17	38.78	42.08
300	54.49	44.52	34.44
400	52.20	41.73	38.61
500	71.47	62.66	47.41
600	68.69	59.15	31.21
700	50.58	24.59	23.15
800	61.01	45.49	43.89
900	76.09	54.57	50.37
1000	55.38	47.64	33.47
Average	58.68	45.85	37.75

texture and $64 \times 64 \times 16$ bytes for info texture. Thus, the proposed technique requires 6.4 times less memory.

Performance-wise comparison between [15] with cluster features (centers and velocities) calculation on the CPU and the proposed method is represented in Tables 5 and 6. In Tables 5 and 6, also a modified version of the proposed method which bypasses the second pass and calculate the cluster features on the CPU is given. The reason for the latter is to demonstrate the benefits of calculating cluster features on the GPU. We tested these three clustering approaches using the previously presented test scenarios with 500 and 1000 virtual agents and results showing the average time (in milliseconds) required by these approaches are given.

5.2.2. Performance evaluation of navigation stage

In order to assess the performance of the proposed system in terms of navigation of virtual agents, we used the experiment scenarios mentioned above. During the first and the second test scenarios, the average time required per agent to complete a single behavior simulation step is measured. The ratios of the differences between clustered C_{sim} and unclustered U_{sim} simulations with respect to unclustered U_{sim} simulations $((C_{sim}-U_{sim})/U_{sim})$ are given in Tables 7 and 8 respectively. According to the results of Scenario 1, there is an extra computational burden of 45–50% on average. However, in Scenario 2, the average computational overhead is only about 7–9% because, in Scenario 2, agent RVO has to consider obstacles on the environment. Thereby, overall calculation time of the agent RVO increases and the cluster calculation overhead relatively decreases. On the other hand, negative numbers in

Table 8
Scenario 2—average performance overheads ratios (%) for each technique with respect to unclustered simulation.

Number of agents	RVO	Reynolds	Social Forces
100	9.28	4.32	0.65
200	21.82	13.49	3.33
300	5.96	−1.15	−4.21
400	−0.55	−8.08	−14.74
500	34.98	15.88	2.80
600	22.70	8.59	29.05
700	30.33	22.33	24.89
800	20.56	11.05	2.80
900	−5.52	−5.52	−14.11
1000	22.48	12.17	2.99
Average	16.20	7.31	3.34

Table 8 indicate hybrid system doing less work overall. The reason for that is unclustered simulation produces more lock-ups (especially in between the obstacles) therefore agents require checking more agents (and obstacles) in unclustered RVO. In addition, it should be noted that given scenarios are extreme situations that stress test the proposed system since almost every cluster and individual meet at the center of the test environment thus, each individual determines cluster guidance force for almost every cluster.

5.2.3. Overall performance

With respect to overall performance (clustering, navigation, animation and rendering), the proposed system is tested by simulating the previously presented scenarios, rendering 500 and 1000 texture-mapped characters, each of which consist of 5028 polygons (as shown in Fig. 10). We used HALCA (Hardware Accelerated Library for Character Animation) [23] to animate the characters on the GPU. Thus, while the CPU is performing the locomotion calculations, we also keep the GPU quite busy during the simulation to evaluate the performance of the proposed technique on a highly loaded system. The detailed performance results of both simulations are shown in Fig. 9. The results in Fig. 9 suggest that the proposed techniques' overhead is about 6–7% with respect to unclustered simulations, and that this percentage is not affected by the density (500 or 1000 agents) of the simulation. Also, as can be observed in Fig. 9, the Social Forces based model has the least overhead among the proposed models (albeit the difference is very small). One disadvantage of the proposed model is that when the agents in the simulation get together at the center of the simulation area (extreme density), the frame rate of the unclustered simulation is more consistent than the proposed techniques.

6. Conclusions

In this study, we propose a hybrid crowd simulation model. The model uses a microscopic locomotion model with additional macroscopic information (clusters). By using geometry shader, fragment shader and hardware accelerated blending, our method not only determines the agent clusters but also determines their centers and average velocities, which are then used in agent locomotion. One other advantage of our method is that the data do not have to return to the CPU memory between the cluster determination and the cluster center and average velocity calculation. Thus, our method minimizes the costly GPU-to-CPU data transfer process. Consequently, we show that by using global cluster information, agents navigate the environment in a more organized fashion, similar to a macroscopic simulation.

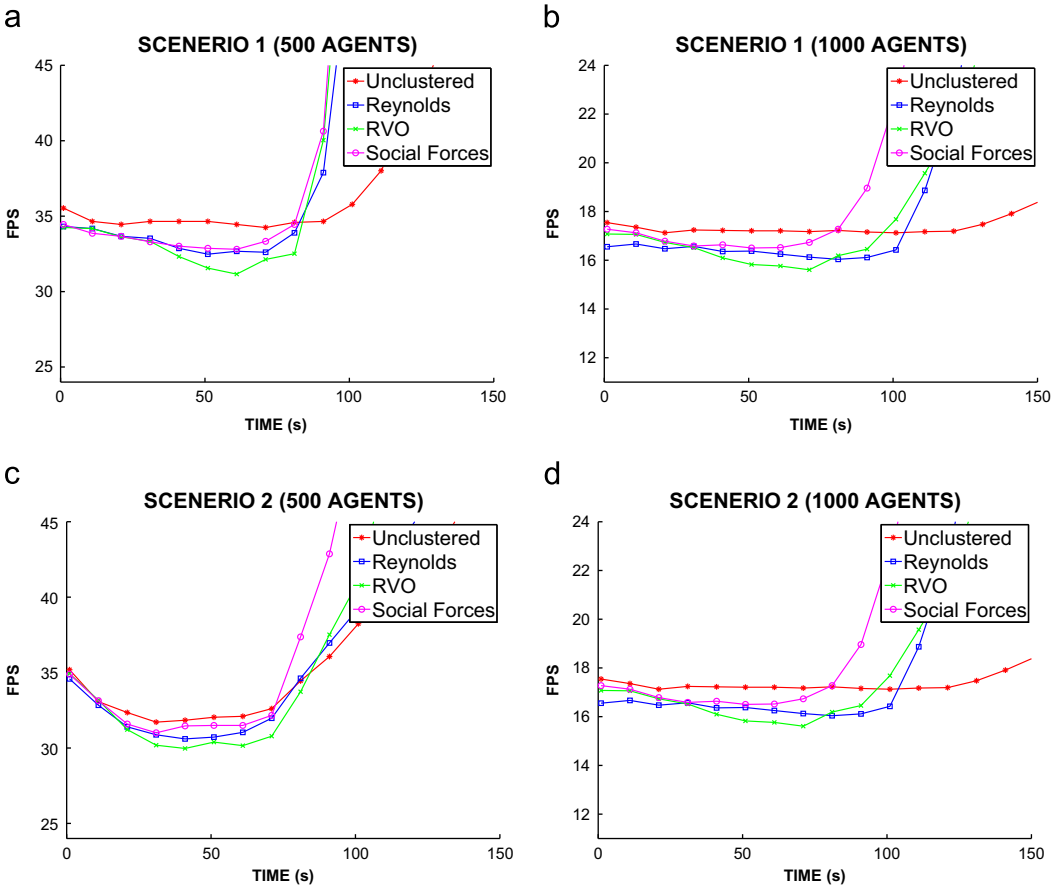

Fig. 9. Performance results for Scenarios 1 and 2 for 500 and 1000 fully rendered agents. As the simulation progresses, the number of simulated agents decreases, thus the frame rate increases for each technique.

Fig. 10. A screen shot from the real-time simulation.

We believe that not only the agent based simulations but also the continuum based simulations can benefit from the dynamic cluster information. Therefore, in the future, to increase efficiency, we will investigate ways to incorporate the cluster information into a macroscopic simulation model.

Appendix A. Supplementary data

Supplementary data associated with this article can be found in the online version at http://dx.doi.org.10.1016/j.cag.2013.05.006.

References

[1] van den Berg J, Guy S, Lin M, Manocha D. Reciprocal n-body collision avoidance. In: 14th international symposium on robotics research; 2009.

[2] Ryder G, Day AM. Survey of real-time rendering techniques for crowds. Comput Graph Forum 2005;24:203–15.

[3] Thalmann D, O'Sullivan C, Ciechomski P, Dobbyn S. Populating virtual environments with crowds. In: Eurographics tutorials, Vienna, Austria; 2006.

[4] Pelechano N, Allbeck J, Badler N. Virtual crowds: methods, simulation, and control. Morgan & Claypool Publishers; 2008.

[5] Thalmann D, Grillon H, Maim J, Yersin B. Challenges in crowd simulation. In: International conference on cyberworlds. Bradford, UK: IEEE Computer Society; 2009. p. 1–12.

[6] Shao W, Terzopoulos D. Autonomous pedestrians. In: Proceedings of the ACM SIGGRAPH/eurographics symposium on computer animation; 2005. p. 19–28.

[7] Maim J, Yersin B, Pettre J, Thalmann D. Yaq: an architecture for real-time navigation and rendering of varied crowds. IEEE Comput Graph Appl 2009;29:44–53.

[8] Paris S, Donikian S. Activity-driven populace: a cognitive approach to crowd simulation. IEEE Comput Graph Appl 2009;29:34–43.

[9] Reynolds CW. Steering behaviors for autonomous characters. In: Game developers conference '09; 2009. p. 763–82.

[10] Helbing D, Molnar P. Social force model for pedestrian dynamics. Phys Rev 1995;51:4282–6.

[11] Treuille A, Cooper S, Popovic Z. Continuum crowds. In: SIGGRAPH, Boston, Massachusetts USA; 2006. p. 1160–8.

[12] Narain R, Golas A, Curtis S, Lin M. Aggregate dynamics for dense crowd simulation. ACM Trans Graph 2009;28:122:1–8.

[13] Guy S, Chhugani J, Curtis S, Dubey P, Lin M, Manocha D. Pledestrians: a least-effort approach to crowd simulation. In: ACM SIGGRAPH/eurographics symposium on computer animation. Aire-la-Ville, Switzerland: Eurographics Association; 2010. p. 119–28.

[14] Golas A, Narain R, Lin M., Hybrid long-range collision avoidance for crowd simulation. In: Proceedings of the ACM SIGGRAPH symposium on interactive 3D graphics and games, I3D' 13. ACM; New York, NY, USA: 2013. p. 29–36.

[15] Haciomeroglu M, Ozcan CY, Barut O, Seckin L, Sever H. Hardware-accelerated dynamic clustering of virtual? Crowd members, Comput. Anim. Virtual Worlds 2013;24:143–53.

[16] Rudomín I, Millán E, Hernández B. Fragment shaders for agent animation using finite state machines. Simulation Model Pract Theory 2005;13:741–51.

[17] D'Souza RM, Lysenko M, Rahmani K. Sugarscape on steroids: simulating over a million agents at interactive rates. In: Proceedings of Agent 2007; 2007.

[18] Joselli M, Passos E, Zamith M, Clua E, Montenegro A, Feijo B. A neighborhood grid data structure for massive 3d crowd simulation on gpu. In: VIII Brazilian symposium on digital games and entertainment, Rio de Janeiro, Brazil; 2009. p. 121–31.
[19] Hall J, Hart J. Gpu acceleration of iterative clustering. In: ACM workshop on general purpose computing on graphics processors; 2004. p. C-6.
[20] Cao F, Tung AKH, Zhou A. Scalable clustering using graphics processors. In: Advances in web-age information management. Lecture notes in computer science, vol. 4016; 2006. p. 372–84.
[21] Shalom SA, Dash M, Tue M. Efficient k-means clustering using accelerated graphics processors. In: 10th international conference on data warehousing and knowledge discovery. Berlin, Heidelberg, Germany: Springer-Verlag; 2008. p. 166–75.
[22] Ward JH. Hierarchical grouping to optimize an objective function. J Am Stat Assoc 1963;58:236–44.
[23] Spanlang B. HALCA hardware accelerated library for character animation. Technical Report; 2009.