A Flocking Boids Simulation and Optimization Structure for Mobile Multicore Architectures

Mark Joselli, UFF, IC, Medialab
Erick Baptista Passos, IFPI, LIMS
Jose Ricardo Silva Junior, UFF, IC, Medialab
Marcelo Zamith, UFF, IC, Medialab
Esteban Clua, UFF, IC, Medialab

Eduardo Soluri
Nullpointer Tecnologias
http://www.nullpointer.com.br

Abstract

During the past few years, mobile phones and other mobile devices have gone from simple phone and messaging devices to high end smartphones with serious computing capabilities. Nowadays, most of these devices are equipped with multicore processors like dual- core CPUs and GPUs, which are designed for both low power consumption and high performance computation. Moreover, most devices still lack libraries for generic multicore computing usage, such as CUDA or OpenCL. However, computing certain specific kind of tasks in these mobile GPUs, and other available multicores processors, may be faster and much more efficient than their single threaded CPU counterparts. In this work, we present a novel approach for flocking boids simulation based on the Android renderscript API. We describe and implement a custom neighborhood grid, and present results with a simple game based on this platform.

Keywords:: Mobile, Mobile Multicore Computing, Flocking Boids, RenderScript, Android, Crowd Simulation

Author's Contact:

mjoselli@ic.uff.br
erickpassos@ifpi.edu.br
jricardo@ic.uff.br
mzamith@ic.uff.br
esteban@ic.uff.br
esoluri@nullpointer.com.br

1 Introduction

In a typical natural environments it is common to find a huge number of animals, plants and small dynamic particles [Ricardo da Silva Junior et al. 2012]. This is also the case in other densely populated systems, such as sport arenas, communities of ants, bees and other insects, or even streams of blood cells in our circulatory system. Computer simulations of these systems usually present a very limited number of independent entities, mostly with very predictable behavior. There are several approaches that aim to include more realistic behavioral models for crowd simulation such [Reynolds 1987; Musse and Thalmann 1997; Shao and Terzopoulos 2005; Rodrigues et al. 2010; Pelechano et al. 2007; Treuille et al. 2006]. All these models are based on the flocking boids approach [Reynolds 1987], which also fundaments this work. While high end games traditionally use crowd environments, due its high end hardware resources, mobile games avoid them.

Algorithms for flocking simulation are driven by the need to avoid the $O(n^2)$ complexity of the proximity queries between entities. Several approaches have been proposed to cope with this issue [Reynolds 2000; Chiara et al. 2004; Courty and Musse 2005] but none of them has reached an ideal level of scalability. As far as we know, no work until the present date has proposed a real time simulation of more than just a few hundreds of complex entities interacting with each other on a mobile device.

Crowd simulation are now appearing frequently on computer games, like Gran Theft Auto IV [North 2008], and digital films, like trilogy of The Lord of the Rings [Aitken et al. 2004]. Typical examples of the use of crowd simulation are the simulation of the behavior of herbs of animals [Reynolds 1987], people walking on the street [van den Berg et al. 2008], soldiers fighting in a battle [Jin et al. 2007] and spectators watching a performance [nVidia 2008]. This work also presents a simple game prototype, using the boids emergent behavior.

Digital games are defined as real-time multimedia applications that have time constraints to run their tasks [Joselli et al. 2010]. If the game is not able to execute its processing under some time threshold, it will fail [Joselli and Clua 2009a]. Mobile games are also real-time multimedia application that runs on mobile phones that have time constraints and many others characteristics [Joselli et al. 2012a], when compared to PC or console games, like: hardware (processing power and screen size); user input, (buttons, voice, touch screen and accelerometers); and a big diversity of operating systems, like Android, iPhone OS, Symbian and Windows Mobile[Joselli and Clua 2009b].

Mobile devices are a growing market [Koivisto 2006]. Devices powered with Android have 60% of the sale market share in the first quarter of this year in the USA, according with [CNET 2012]. Also the use of the internet on such devices are gaining importance, since its has been doubling even year [GlobalStats 2012]. These are important motivations for game developers and designer to create blockbusters and high end games.

Google introduced in the Honeycomb version of Android the Renderscript API (application programming interface) [Android 2012]. Renderscript is an API for achieving better performance on Android phones and tablets. Using this API, applications can use the same code to run on different hardware architectures like different CPUs (Central Processing Unity), ARM (Advanced RISC Machine) v5, ARM v7, and X86, GPUs (Graphic Processing Unit) and DSPs (Digital Signal Processors). The API decides which processor will run the code on the device at runtime, choosing the best processor for the available code. This work presents a novel modeling of flocking boids data structures suitable for this new architecture and compares it to the traditional brute force algorithm. As far as the authors knows, this is the first flocking boid simulation that uses this kind of approach.

Most of the research on flocking boids simulations tries to avoid the high complexity of proximity queries by applying some form of spatial subdivision to the environment and classifying entities among the cells based on their position. Since the Renderscript is very new and have some scatter constraints, there are lack of spatial subdivisions techniques implemented in this technology, so most works uses the brute force algorithm, which has a $O(n^2)$ complexity. In this paper, instead of using a similar approach, we propose a novel simulation architecture that maintains entities into another kind of proximity based data structure, which we call neighborhood grid. In this data structure, each cell now fits only one entity and does not directly represent a discrete spatial subdivision. The neighborhood grid is an approximate representation of the system of neighborhoods on the environment, which maps the N-dimensional environment to a discrete map (lattice) with N dimensions, so that entities that are close in a neighborhood sense, appear close to each other in the map. Another approach is to think of it as a multi-dimensional compression of the environment that still keeps the original position information of all entities.

The entities are simulated and sorted as Cellular Automata with Extended Moore Neighborhood [Sarkar 2000] over the neighborhood grid, which is an ideal case for the memory model of multicores architectures. We argue and show that this approximate simulation technique brings a new bound to crowd simulation performance, maintaining the believability for entertainment contexts. The high performance and scalability are achieved by a very low parallel complexity of the model. To keep the neighborhood grid aligned this work shows a implementation of a partial sorting mechanism, a parallel bubble sort, implemented with the renderscript.

To illustrate and evaluate our proposal, we implement a traditional emergent behavior model of flocking boids [Reynolds 1987] that has a minimum speedup of 2.14 over the tradition brute force method, with similar visual experience. The architecture can be further extended to any other simulation model that rely on dynamic autonomous entities and neighborhood information. Also a game prototype, based on the boids rules are developed and presented.

Summarizing, this work is an extension of the work [Joselli et al. 2009; Passos et al. 2010], with the following enhancements, which are the main contributions of these paper:

- Modeling of the flocking boids architecture on a mobile device;
- Adaptation of the architecture and data structures for Renderscript API;
- Implementation of a game based on the boids rules.

The paper is organized as follows: Section 2 discusses the mobile GPUs and the renderscript API. Section 3 presents the related work on crowd simulation and on mobile multicore processors. Sections 4 explain the neighborhood grid, the architecture used, the data structures and the simulation steps. Section 5 describes the particular behavior model used to validate the proposed architecture, and the game used on tests. Section 6 brings the experimental results and analysis of the implemented simulation model. Finally, section 7 concludes the paper with a discussion on future work.

2 Mobile Multicore Processing

Multicore architecture are, nowadays, present in home PCs and mobile phone, available in the multicore CPUs and GPUs. GPUs are powerful processors originally dedicated to graphics computation [Joselli et al. 2012b]. GPUs for PCs composed by hundreds of parallel processors, achieving much better performance then modern CPUs in several applications scenarios [Feinbube et al. 2011]. Kepler K10,for instance, can sustain a measured 4.5 TFLOPS/s against 60 GFLOPS/s of its contemporary CPU processors [Aila et al. 2012]. The GPU can be used on the PC as a generic processor to process data and deal with computationally intensive tasks, through development of elaborate architectures such as CUDA (Compute Unified Device Architecture) [nVidia 2009] and OpenCL[Group 2009] . These architectures facilitates the use of the GPU computing for generic processing, and can be seen applied in many different scenarios, like: geologic [Kadlec et al. 2009], medical [Muyan-Ozcelik et al. 2008] and computer vision [TunaCode 2010].

On mobile devices, the GPU is much less capable and powerful [Akenine-Moller and Strom 2008], and is typically integrated into the mobile processor system-on-a-chip (SoC), which also consists of one or several CPUs, DSP (digital system processor), and other available mobile-specific accelerators. This embedded GPU does not have a memory specific for it, having to share the system bus, with the others processors for accessing the memory. Consequently the memory bandwidth is also much lower when compared to the desktops GPUs [Cheng and Wang 2011].

Currently, mobile GPUs emphasis more on lower power consumption [Therdsteerasukdi et al. 2012] than performance. Some of these currently available Gpus devices are the Qualcomm's Adreno 200 GPU, the TI's PowerVR SGX 530/535 GPU and the nVidia Tegra2 GPU.

Normally, most works that uses mobile for parallel processing, deals with the use of the GPU for generic processing with the OpenGL ES [Munshi et al. 2008] programable shaders, the vertex and fragment shader, as the programming interface [Kim et al. 2007]. The disadvantage of some approaches is the traditional shader languages limitations (such as scatter memory operations, i.e. indexed write array operations), and offering others features that are not even implemented on those languages (such as integer data operands like bit-wise logical operations AND, OR, XOR, NOT and bit-shifts) [Owens et al. 2007]. Some of these disadvantages are also present in the rendersript API, like the limitation of scatter memory operations.

2.1 The Renderscript API

Renderscript is a new software development kit and API for Android firstly introduced by Google in the Honeycomb version of Android. Renderscript is an API for high-performance graphics processing on Android phones and tablets. It is used for fast 3D rendering and computing processing, having similar paradigm as GPU computing libraries and frameworks [Huang et al. 2011]. The main goal of Renderscript API is to bring a lower level, higher performance API to Android developers, in order to achieve better performance in visual animations and simulations [Guihot 2012].

Renderscript code is compiled on the device at runtime, so the developer do not need to recompile the application for different processor types, making more easy its usage [Ostrander 2012]. Its language is an extension of the C99 language that is translated to an intermediate code at compile time, and then to machine code at runtime. The API scale the generated code to the amount of processing cores available on the device. The decision of choosing which processor will run the code is made on the device at runtime, being completely transparent for the developer. Normally simple scripts will be able to run on the available GPUs, while more complex scripts will run on the CPU. The CPU is also a fallback, so that if none other available suitable device, it will run the code.

All the tasks implemented in Renderscript are automatic portable for parallel processing on the available processors of the device, like the CPU, GPU and even DSP. Renderscript is specially useful for apps that do image processing, mathematical modeling, or any operations that require lots of mathematical computation, similar to GPU computing paradigm. The main use of renderscript is to gain performance in critical code where the traditional Android framework and OpenGL ES APIs are not fast enough.

The Renderscript is composed of two APIS: a computing API (responsible for processing the computation), and a rendering API (responsible for the tenderization of the scene, working together with OpenGl Es 2.0). The Renderscript code is called from a Android Activity inside the virtual machine. If the code can execute on a GPU or on a multi-core CPU, it may be assigned to run on that. The script runs asynchronously and sends its results back into the Virtual Machine.

3 Related Work

There are not much works on the literature that deals with the use of the mobile on multicore processors. Most of the works deals with image processing, using the GPU for generic processing. In the works [Singhal et al. 2010a] , [Singhal et al. 2011] and [Singhal et al. 2010b] some image processing algorithms where designed and implemented on handheld device using OpenGL ES 2.0. In [Lopez et al. 2011] a mobile-GPU implementation of Local Binary Pattern feature extraction is presented, showing a better performance and power consumption when used the CPU together with the GPU.

Also using OpenGL 2.0 for image processing [Cheng and Wang 2011] shows a face recognition algorithm, with a 4.25x speedup and a 3.88x reduction on the total energy consumption. [Jeong et al. 2009] presents an implementation of GPU-based window system on top of EGL and OpenVG. Also the openCV [Pulli et al. 2012] is a library for computer vision, includes some new and experimental features for the mobile devices. These works are particularly important since they show that the use of the mobile GPU is faster and have low power consumption, and our work also uses the Gpu for its processing.

Nah et al. [Nah et al. 2010] shows OpenGL ES-based CPU-GPU hybrid ray tracer for mobile devices, using Kd-trees. In [M. et al. 2009], a system for building document mosaic images from selected video frames on mobile phones using the GPU for accelerating its processing is presented. Also a image deformation implementation with a misc of ARM-Linux and OpenGL ES for mobile device is presented in [Hu et al. 2011].

There are also some works [Barboza et al. 2010; Zamith et al. 2011; OnLive 2012] that uses cloud-computing for distribution of the processing over the cloud for mobile real time simulation and games. These approaches tend to rely on the network for these distribution, which can be very unreliable and slow using the mobile phones carrier.

There are no works on the literature that deals with the use of renderscript, like this one does. But the Android SDK [Android 2012] makes available a series of sample codes, for building simple animations based on particle systems, like a fountain and a brute force physics simulation that can render and process up to 900 interacting particles.

The first known agent-based simulation for groups of interacting animals is the work proposed by Craig Reynolds [Reynolds 1987], in which he presented a distributed behavioral model to perform this task. His model is similar to a particle system where each individual is independently simulated and acts accordantly to its observation of the environment, including physical rules such as gravity, and influences of other individuals perceived in the surroundings. The main drawback of the proposed approach is the $O(n^2)$ complexity of the traversal algorithm needed to perform the proximity tests for each pair of individuals. This was such an issue at the time that the simulation had to be run as an offline batch process, even for a limited number of individuals. In order to cope with this limitation, the author suggested the use of spatial hashing. This work also introduced the term *boid* (abbreviation for birdoid) that has been used to designate generic simulated flocking creatures ever since.

Musse and Thalmman [Musse and Thalmann 1997] propose a more complex modeling of human motion based on internal goal-oriented parameters and the group interactions that emerge from the simulation, taking into account sociological aspects of human relations. Others include psychological effects [Pelechano et al. 2007], social forces [Cordeiro et al. 2005] or even knowledge and learning aspects [Funge et al. 1999]. Shao and Terzopoulos [Shao and Terzopoulos 2005] extend the latter including path planning and visibility for pedestrians. It is important to mention that these proposals are mainly focused on the correctness aspects of behavior modeling. The data structures and algorithms used by these works are not suitable for real-time simulation of very large crowds, which is one of the goals of this work.

Reynolds further enhanced his behavioral model to include more complex rules and to achieve the desired interactive performance by the use of spatial hashing [Reynolds 2000; Reynolds 1999]. This implementation could simulate up to 280 boids at 60 fps in a Playstation 2 hardware. Also the work by Silva et al. [Silva et al. 2008] implement a similar work, but it focus on the optimization of the algorithm by doing occlusion based on the vision of the boids. By using the spatial hash to classify the boids into a grid, the proximity query algorithm could be performed against a reduced number of pairs. For each boid, only those inside the same grid cell and at adjacent ones, depending on its position, were considered. This strategy leads to a sequential complexity that is closer to $O(n)$. This complexity, however, is highly dependent on the maximum density of each grid cell, which can be very high if the simulated environment is large and dense. We remark that the complexity of our neighborhood grid is not affected by the size of the environment or the distribution of the boids over it.

Quinn et al. [Quinn et al. 2003] used distributed multiprocessors to simulate evacuation scenarios up to 10,000 individuals at 45 fps on a cluster connected by a gigabit switch. More recently, a similar spatial hashing data-structure was used by Reynolds [Reynolds 2006] to render up to 15,000 boids in Playstation 3 hardware at interactive framerates, but with a reduced simulation frame rate of around 10 fps. Due to the distributed memory of both architectures, it is necessary to copy compact versions of the buckets/cells of boids to the individual parallel processors before the simulation step could run, copying them back at the end of it to perform the rendering, which leads to a potential performance bottleneck for larger sets of boids. This issue is evidenced in [Steed and Abou-Haidar 2003], where the authors span the crowd simulation over several network servers and conclude that moving individuals between servers is an expensive operation.

In the works [Passos et al. 2008; Joselli et al. 2009; Passos et al. 2010], which this work extends, is implemented a crowd simulation system on a desktop GPU, using CUDA, where each boid is modeled as a cellular automaton [Sarkar 2000]. This work could achieve the simulation and renderization of up to 1 millions boids at interactive frame rates. The present paper extends that previous work new architecture, the mobile device architecture, with all the unique characteristics and constraints.

4 Proposed Architecture

Individual entities in crowd behavior simulations depend on observations of their surrounding neighbors to decide which actions to take. The straightforward implementation of the neighborhood gathering algorithm has a complexity of $O(n^2)$, for n entities, since it performs at least one proximity query for each entity pair in the crowd. Individuals are autonomous and can move during each frame, which leads to a very computationally intensive task. This implementation is showed in algorithm 1.

Algorithm 1 Brute Force Neighborhood Gathering Algorithm

```
for i=0;i<number of entities;i++ do
  for j=0;j<number of entities;j++ do
    if i ! = j AND distance(i,j) ¡ k then
      Do computation with (i,j)
    end if
  end for
end for
```

Techniques of spatial subdivision have been used to group and sort these entities in order to accelerate the neighborhood finding task. Current implementations are usually based on variations of relatively coarse subdivisions techniques, such as a grid over the considered environment. After each update, all entities have their grid cell index calculated based on their latest locations. But the scatter memory operations on the Renderscript API is still very primitive, lacking the ability to process this kind of structure. So most algorithms that deals with some sort of neighborhood gathering uses the brute force algorithm, which has an $O(n^2)$ complexity factor.

In this work we use another approach for the neighborhood gathering problem. This approach uses a grid data structure, which is called neighborhood grid that is used to store information about all the entities. In this neighborhood grid, each entity is mapped in a individual cell (1:1 mapping) accordingly to its spatial location, so that entities that are close in a neighborhood sense, appear close to each other in the grid. In order to keep the neighborhood grid mapped accordingly to the spatial location, a sorting mechanism is needed. To fulfill that need, we do a partial sort at each step the structure to keep the relations aligned.

In order to fully function on the mobile device, this simulation architecture is divided in four different ambients:

- the Android framework, where the application is created, and the renderscript context is also created. Also, this ambient is responsible for gathering the inputs and to sent it to the computing renderscript;
- the computing renderscript is where the variables for the simulation are created, the call for the renderscript computing engine are made and the sort the entities is done;

- the renderscript computing engine will process the behavior of the scene distributing its process among the available processors;
- OpenGL: will render all the objects, applying shaders and visual effects to them.

This architecture is illustrated on figure 1.

The following subsections describe the architecture. In the next subsection the neighborhood grid is explained. The role of sorting the grid are also explained in follow subsection.

4.1 The Neighborhood Grid

The proposed architecture was manly built using the renderscript API, in order to achieve better performance by accessing the available multicores processors of the mobile device. Structs for the information of each entity, which consists of: position (a vector, representing the position of the entity), speed (a vector for storing the orientation and velocity in a single structure) and type (an integer that can be used to differentiate entity classes).

All the information about the entities are stored in 3D arrays (grid), where each position holds the entire data for an individual entity, which contain the struct with all the information of the entity. In this data structure, each cell fits only one entity. Figure 2 illustrates how a randomly distributed set of entities would be arranged in the neighborhood grid when correctly sorted viewed from a top-down camera.

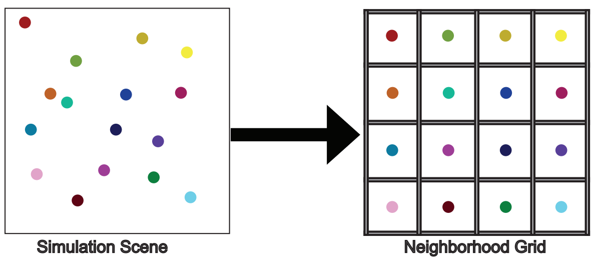


Figure 2: *Construction of the Neighborhood grid in a top-down camera.*

In this work we use a extended form of neighborhood gathering that is known as Extended Moore Neighborhood [Sarkar 2000] in the Cellular Automata theory. The algorithm for gathering such a neighborhood can be seen at Algorithm 2. This algorithm takes as input the radius of the neighborhood and the neighborhood grid, the array with the stored particles.

Algorithm 2 Extended Moore Neighborhood Gathering Algorithm

```
for z=-radius;z<=radius;z++ do
  for y=-radius;y<=radius;y++ do
    for x=-radius;x<=radius;x++ do
      if (x != 0 OR y != 0 OR z!=0) then
        Do computation with (Grid[indexZ+z][indexY+y][indexX+x])
      end if
    end for
  end for
end for
```

Figure 3 illustrates this structure in our 3D matrix holding arbitrary information about the individual entities. To reduce the cost of proximity queries, each entity will only gather information about the entities surrounding its cell, based on a constant radius. In the example of Figure 3, this radius is 1, so the entity represented at cell in gray would have access to the highlighted surrounding cells/entities in green.

This kind of spatial data structure and extremely regular information gathering enables a good prediction of the performance, since

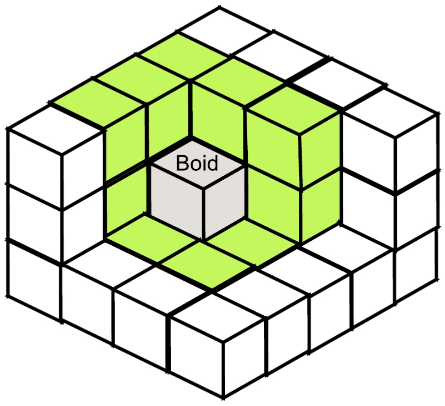


Figure 3: *Example of the neighborhood grid with radius = 1.*

the number of proximity queries will always be constant over the simulation. This happens because instead of making these proximity queries over all entities inside a coarse grid/bucket/cell (with variable quantity), such as in spatial subdivision implementations, each entity would query only a fixed number of surrounding individual neighbors. However, this grid has to be sorted continually in such a way that those entities which are neighbors in geometric space are stored in individual cells that are close to each other. This guarantees that each entity should gather information about its closest neighbors. Depending on the simulation (and the sorting step), some misalignment may occur over the data structure, causing that some of the neighbor entities are missed by the gathering step. However, the larger the Moore radius is, less likely it is to happen such issue, which we could observe during the experiments.

4.2 Sorting Pass

The position information of each entity is used to perform a lexicographical sort based on the three dimensions of this vector. The goal is to store in the closer-bottom-leftmost cell of the grid the entity with the smaller values for Z, Y and X, and in the far-top-rightmost cell the entity with highest values of Z, Y and X respectively. Using these three values to sort the matrix, the farthest lines will be filled with the entities with the higher values of Z while the top lines will be filled with the entities with higher values of Y and the right columns will store those with higher values for X and so on. This kind of sorting provides for the approximate neighborhood query that is optimal in terms of data locality.

When performing a sorting over a one dimension array of float point values, the goal is that given an array **A**, the following rule must apply at the end:

- $\forall A[i] \in \mathbf{A},\ i > 0 \Rightarrow A[i\text{-}1] \leq A[i]$.

The architecture is independent of the sorting algorithm used, as long as the rules above are always, eventually or even partially achieved during simulation, depending on the desired neighborhood precision. In this work, we have used a parallel implementation of a partial bubble sort algorithm. In our test case, it is sufficient to run only one partial bubble sort pass for each simulation update because we initialize the neighborhood grid in an ordered state and the flocking nature of the simulation algorithm implies that the entities do not overlap positions frequently. In practice, this means that after very few simulation steps, the grid correctly represents the proximity relations of the entities. Depending on the simulation being performed, it may be necessary to perform a complete sorting at each update step. In this case, it is recommended a sorting algorithm with better worst case complexity, such as a parallel merge or radix sort. However, for all scenarios experimented in our work, the (incomplete) sort was enough to sustain an approximately correct simulation, with no noticeable visual artifacts

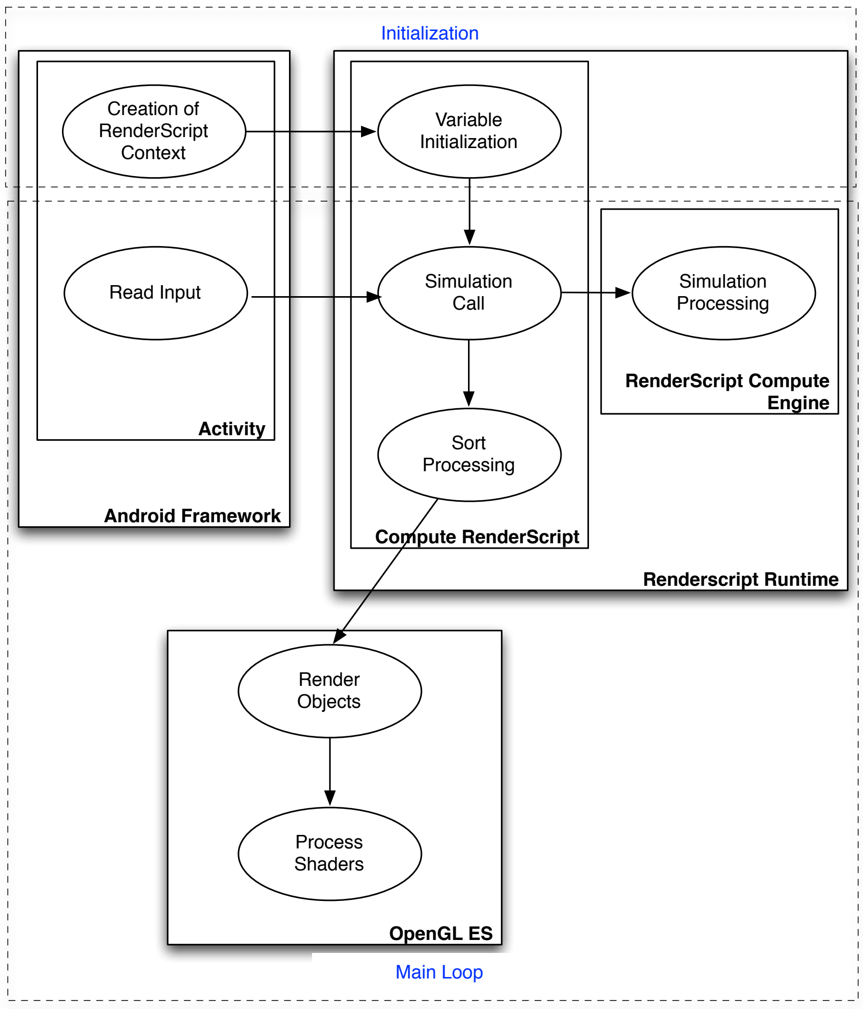


Figure 1: *Architecture Overview.*

5 Flocking Boids

For the purpose of this work, we choose to validate the proposed technique by implementing a well known distributed simulation algorithm called, flocking boids [Reynolds 1987]. This is a good algorithm to use because of its good visual results, proximity to real world behavior observation of animals and understandability. The implementation of the flocking boids model using our algorithm enables a real time simulation of up to thirty two thousands animals of several species, with a corresponding visual feedback. The number of different species is limited only by the number of animals in the simulation.

Our model simulates a crowd of animals interacting with each other and avoiding random obstacles around the space. This simulation can be used to represent from small bird flocks to huge and complex terrestrial animal groups or either thousand of hundreds of different cells in a living system. Boids from the same type (representing the species) try to form groups and avoid staying close to the other type of species. The number of simulated boids and types is limited only by technology but, as demonstrated in the next section, our method scales very well due to the data structures used. In this section we focus at the extension of the concepts of cellular automata in the simulation step, in order to represent emergent animal behavior.

To achieve a believable simulation we try to mimic what is observable in nature: many animal behaviors resemble that of state machines and cellular automata, where a combination of internal and external factors defines which actions are taken and how they are made. A state machine is used to decide which actions are taken. The actions themselves performed by a cellular automaton algorithm. With this approach, internal state is represented by the boid type and external ones corresponds to the visible neighbors, depending from where the boid is looking at (direction), and their relative distances.

Based on these ideas, our simulation algorithm uses internal and external states to compute these influences for each boid: Flocking (grouping, repulsion and direction following); leader following; and other boid types repulsion (used also for obstacle avoidance). Additionally, there are multiplier factors which dictate how each influence type may get blended to another, in each step. In order to enable a richer simulation, these factors are stored independent for each type of boid in separate arrays.

The remainder of the section is divided by each behavior rule of the flocking. also the last subsection is dedicated for the Boidoid game, which is a game based on the flocking boids rules.

5.1 Vision

In nature, each animal species has a particular eye placement, evolved based on its survival needs such as focusing on a prey or covering a larger field of view to detect predators. To mimic this fact, our boids have a limited field of view, parameterized by an angle. Obstacles and other boids outside this field of view are not considered in the simulation. Figure 4 shows a comprehensible representation of this field of view.

When two boids are very close to each other, up to collide, corresponds to a special case where a boid takes into account a neighbor even if it is outside of the its field of view. If collisions where

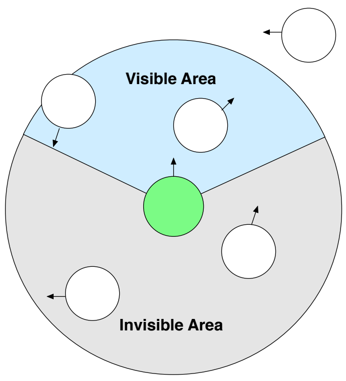


Figure 4: *The visual field of a boid*

allowed to happen, the simulation could become unstable since neighbor boids coming from behind would suddenly appear in front of another. It is possible to think of this as a collision detection for a prevention system, having the same effect as a movement made by animals that, even not seeing each other, would have gotten into a sudden contact.

5.2 Flocking Behavior

A boid keeps on moving by watching his visible neighbors and deciding what direction to take next. Each neighbor influences this direction in different conflicting manners, depending on its type and distance from the simulated boid. From neighbors of the same type, the simulated one receives three simultaneous influences: grouping, repulsion and direction following.

5.2.1 Grouping Influence

By grouping we mean the tendency that animals from the same species have to keep forming relatively tight groups. To simulate this behavior we compute the group center position by averaging the positions of all visible neighbors of the same type as the one being simulated. This grouping influence will be multiplied by a grouping factor, unique for each type, and by the distance from the centre. The last factor will make the influence stronger to boids that are far from the group. Figure 5 illustrates grouping and repulsion influences.

5.2.2 Repulsion Influence

If only the grouping influence was taken into account, boids would tend to form very dense groups, resulting in very frequent collisions, not representing what we see in nature. To balance this grouping tendency a collision avoidance influence is computed. For each simulated boid, the relative distance to its neighbors is computed and divided by its length. This weighted vector is then multiplied by a specified repulsion factor and added as an influence to the desired motion vector. One can notice that the parameterized factors of both the grouping and distance influences play a major role in determining the density of the groups, since one cancels each other at a certain distance when equilibrium is reached between them.

5.2.3 Direction Following Influence

Besides the tendency of forming groups, animals also tend to follow the same direction as its companions. To achieve this behavior we compute another influence every time a boid sees a neighbor of the same type. This influence is represented by the current velocity/direction followed by the neighbor. Figure 6 exemplifies this influence.

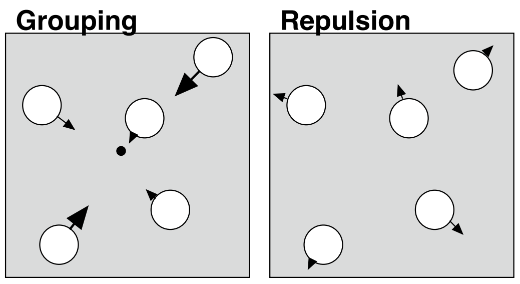


Figure 5: *Grouping and repulsion influences*

5.3 Leader Following

Besides from recognizing its neighbors of the same type and trying to move as a group, each type may have a leader to follow. Normal boids, when see the leader, have a stronger desire to follow it, represented by a larger multiplier factor, that gets blended with the other computed influences. Each leaders is simulated at the same time as normal boids but also being identified as such and acting accordingly. However, the movement of this leader is not driven by the desire to keep grouping, but only trying to reach a desired location and avoiding obstacles and other boid groups.

Inside the data structures, the leaders are represented as normal boids. There is a small auxiliary array keeping the current matrix index exclusively for the leaders of each boid type. The array size is the number of different boid types. Element n of this array contains the cell index of the leader for the boids of type *t*. To be correct along the time, this array must be updated by the sorting pass if any of the leaders change its cell.

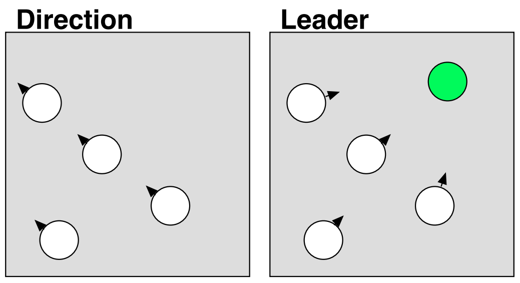


Figure 6: *Direction and leader following*

During the simulation step, for each boid the leaders array value for its type is fetched and the value returned identifies the leader index inside the matrixes. If the returned index corresponds to the boid being simulated, it means that corresponds to the group leader and follows to an alternative and more random simulation algorithm. For normal boids, this leader index is used to fetch its position and direction, so that the correct influence can be computed.

5.4 Obstacles, Enemies and Influences Composition

In this work, obstacles are also represented as boids inserted in the same data structures, also being sorted and simulated. To avoid movement during the simulation step, obstacles are initialized with a different type value, and are not simulated. However, if a neighbor of a specific simulated boid happens to be an obstacle, the only influence calculated a is repulsion force. This force is then multiplied by a factor that is stored in the unused direction vector of this still obstacle-boid, enabling the representation of obstacles of arbitrary sizes with a round repulsion field. Neighbors of different types that are not obstacles also have a strong repulsion influence calculated, but the multiplier factor is kept at the simulated boid type, representing an enemy-fearness factor. All calculated influences are added into an acceleration vector that is used to update the position and direction/speed vectors.

5.5 The Design of Boidoid Game

The Boidoid is a massive prototype action game with a top-down 2D perspective. The story behind the game is that the boids world are collapsing, and the boids need to escape by reaching the portals. But the boids do not know that, and they only tries to respect their flocking rules. Its up to the player to save those boids from extinction.

The game play is very simple, in a world there are up to four different boids, which are represented by different colors. All of these boids respect the flocking rules defined in this section. These boids are random distributed on the screen. Also up to four different portals, one for each type of boid, are distributed on the screen, when the boid are inside, they are collected. The player mission is to collect as many boids as they can, in the available time. Every time the player touches the screen, he creates repulsion forces, that can be used to lead boids to the portals.

6 Results

In this work, we implemented and tested the flocking boids case-study and game prototype using the neighborhood grid and also evaluated the rendering of all boids. The rendering consists of a simple primitive, a sphere representing each void, that is bound from the output VBO (vertex buffer object) of the simulation in a vertex shader. A series of screenshots from the game can be seen on Figure 7.

To evaluate the scalability of the architecture, we varied the number of entities/boids being simulated (from 1 hundred to 32 thousands). The Moore neighborhood radius was set to 4 (which was the radius with better visual behavior). At preliminary tests, we observed that the number of different boid types had no observable influence on the performance, so a fixed number of 4 types was used for all experiments. The simulation of the boids behavior were developed using two algorithms: the neighborhood grid and the brute force algorithm, which were also implemented using the renderscript API.

For the tests, we have used a Asus Tranformes TF101, which is a 10.1 inches tablet with an Android 4.03 operating system that has a Nvidia Tegra 2 T20 chipset with a Dual-core 1 GHz Cortex-A9 CPU and a ULP (ultra-low power) GeForce GPU and 1GB RAM memory. Simulations tests with different configurations were performed. The rendering is done, in screen space, through applying a bilateral filter in sphere's normal. To assure that results are consistent, each test was repeated 10 times and the standard deviation of the average times was confirmed to be within 5%.

In Table 1 and Figure7 show the results of different simulation configurations, by varying the number of boids in the scene. In these results, the label FPS represents the *frames per second* which measure a time necessary to update and render the simulation. *Speedup* is defined by the relation $S = \frac{X_1}{Y_2}$, being X_1 the FPS for the Neighborhood Grid and Y_2 the FPS for the Brute Force algorithm.

Table 1: *Scalability of the Simulation.*

# Boids	Brute Force FPS	Neighborhood Grid FPS	Performance Gain
128	115	247	2.14
256	70	170	2.42
512	40	131	3.25
1024	10	117	11.7
2048	3	101	33
4096	0.50	86	172
8192	0.14	57	407
16384	0.03	30	1000
32768	0.01	16	1600

From these results, as expected, the simulation using the method implemented with the neighborhood grid presents a better result than the simulation using the method implemented with the brute force algorithm. These tests also shows that even with more that 32k interactive boids in the scene, the simulation with neighborhood grid can still maintain the interaction, since it maintain the lower bound for interaction [Joselli et al. 2008]. This work has also implemented the Neighborhood Grid using the the Android NDK which shows a speedup of the Renderscript implementation of up to 3 times.

7 Conclusion

In this paper we have shown an extension of a novel technique for simulating emergent behavior of dynamic entities in a densely populated environment. We have extended all of our data structure to mobile architecture in order to make crowd simulation optimized and suitable for mobile devices. We also have implemented a game based on the boids rules. Also, we must remark, that as far as the authors of this work knows, this is the first massive boid simulation in a common mobile device, and also the first work on literature using the renderscript API.

From the tests, we can see that the simulation is much faster when using the neighborhood grid algorithm. Also other simulations could be implemented using the API and the data structure.

References

AILA, T., LAINE, S., AND KARRAS, T. 2012. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June.

AITKEN, M., BUTLER, G., LEMMON, D., SAINDON, E., PETERS, D., AND WILLIAMS, G. 2004. The lord of the rings: the visual effects that brought middle earth to the screen. In *ACM SIGGRAPH 2004*, ACM Press, New York, NY, USA, SIGGRAPH: ACM Special Interest Group on Computer Graphics and Interactive Techniques.

AKENINE-MOLLER, T., AND STROM, J. 2008. Graphics processing units for handhelds. *Proceedings of the IEEE 96*, 5 (may), 779 –789.

ANDROID, G., 2012. Android renderscript. Avalible at: `http://developer.android.com/guide/topics/renderscript/index.html`.

BARBOZA, D. C., JUNIOR, H. L., CLUA, E. W. G., AND REBELLO, V. E. 2010. A simple architecture for digital games on demand using low performance resources under a cloud computing paradigm. *Games and Digital Entertainment, Brazilian Symposium on 0*, 33–39.

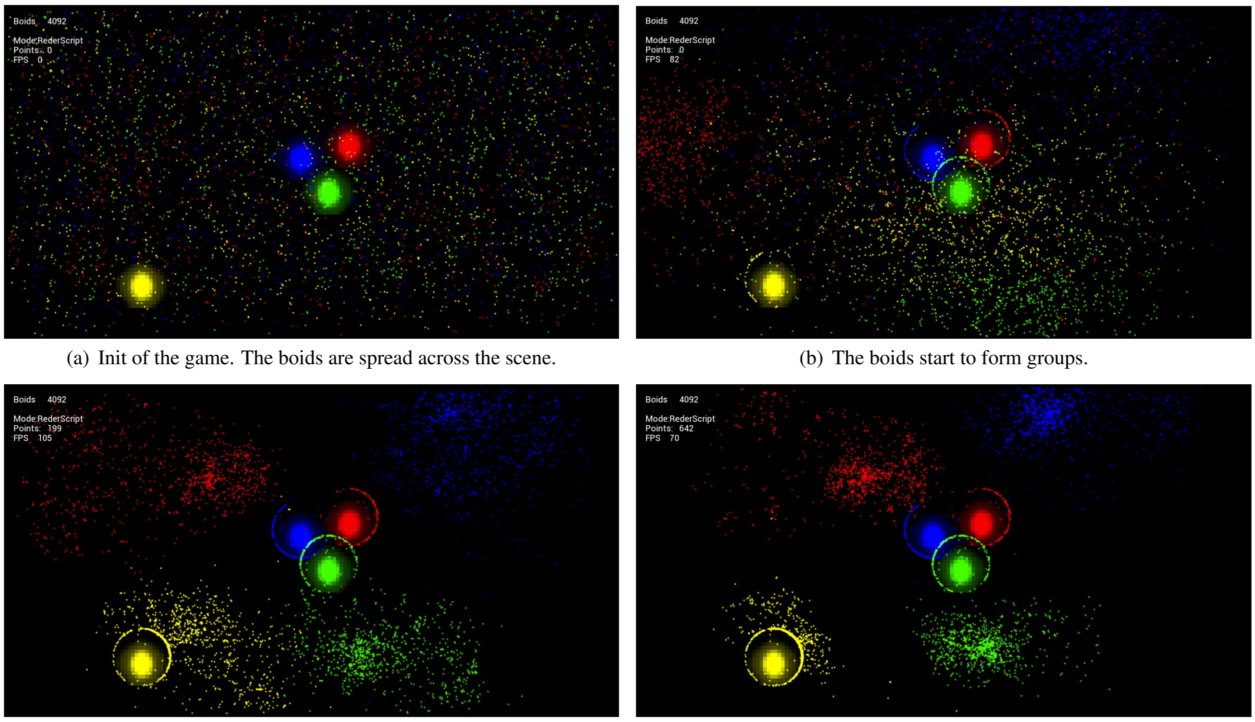
CHENG, K.-T., AND WANG, Y.-C. 2011. Using mobile gpu for general-purpose computing a case study of face recognition on smartphones. In *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, 1 –4.

CHIARA, R. D., ERRA, U., SCARANO, V., AND TATAFIORE, M. 2004. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. In *Vision, Modeling, and Visualization (VMV)*, 233–240.

CNET, 2012. Android reclaims 61 percent of all u.s. smartphone sales. Avalible at: `http://tinyurl.com/cgsszfc`.

CORDEIRO, O. C., BRAUN, A., SILVEIRA, C. B., AND MUSSE, S. R. 2005. Concurrency on social forces simulation model. In *Proceedings of the First International Workshop on Crowd Simulation (V-CROWDS)*, V-CROWDS.

COURTY, N., AND MUSSE, S. R. 2005. Simulation of large crowds in emergency situations including gaseous phenomena. In *CGI '05: Proceedings of the Computer Graphics International 2005*, IEEE Computer Society, Washington, DC, USA, CGI, 206–212.



(a) Init of the game. The boids are spread across the scene.

(b) The boids start to form groups.

(c) The boids groups are formed.

(d) Most of the boids are inside the portals

Figure 7: *Screenshots in difference moments of the game.*

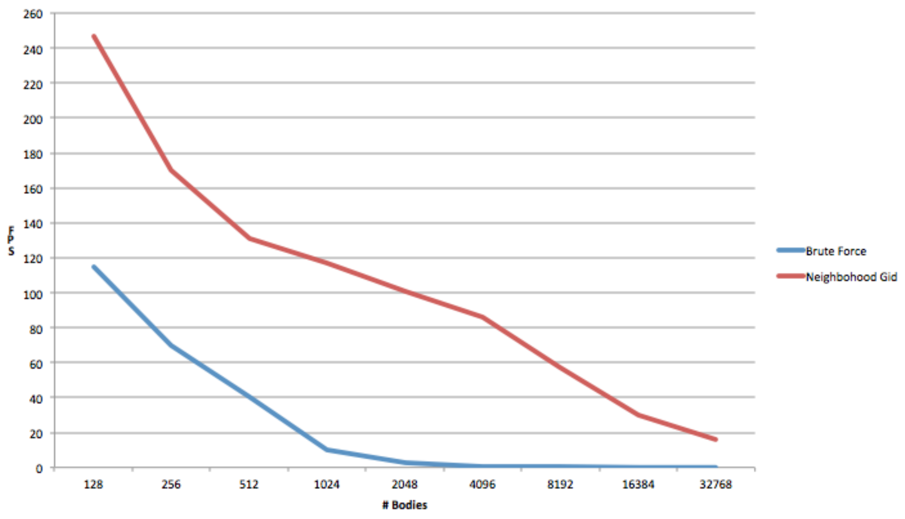


Figure 8: *Performance of the simulation.*

FEINBUBE, F., TRO ANDGER, P., AND POLZE, A. 2011. Joint forces: From multithreaded programming to gpu computing. *Software, IEEE 28*, 1 (jan.-feb.), 51 –57.

FUNGE, J., TU, X., AND TERZOPOULOS, D. 1999. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *Siggraph 1999, Computer Graphics Proceedings*, Addison Wesley Longman, Los Angeles, A. Rockwood, Ed., Siggraph, 29–38.

GLOBALSTATS, 2012. Mobile internet usage is doubling year on year. Avalible at: `http://gs.statcounter.com/press/mobile-internet-usage-is-doubling-year-on-year`.

GROUP, K., 2009. Opencl - the open standard for parallel programming of heterogeneous systems. Avalible at: `http://www.khronos.org/opencl/`.

GUIHOT, H. 2012. *Pro Android Apps Performance Optimization*. Apress.

HU, X., XIA, Z., AND YUAN, Z. 2011. Study on image deformation simulation based on arm linux and opengl es. In *Proceedings of the 2011 International Conference on Intelligence Science and Information Engineering*, IEEE Computer Society, Washington, DC, USA, ISIE '11, 303–306.

HUANG, Y., CHAPMAN, P., AND EVANS, D. 2011. Privacy-preserving applications on smartphones. In *Proceedings of the 6th USENIX conference on Hot topics in security*, USENIX Association, Berkeley, CA, USA, HotSec'11, 4–4.

JEONG, D., KAMALNEET, KIM, N., AND LIM, S. 2009. Gpu-based x server on top of egl and openvg. *Computers in Education, International Conference on 0*, 1–2.

JIN, X., WANG, C. C. L., HUANG, S., AND XU, J. 2007. Interactive control of real-time crowd navigation in virtual environment. In *VRST '07: Proceedings of the 2007 ACM symposium on Virtual reality software and technology*, ACM, New York, NY, USA, 109–112.

JOSELLI, M., AND CLUA, E. 2009. Gpuwars: Design and implementation of a gpgpu game. *Brazilian Symposium on Games and Digital Entertainment*, 132–140.

JOSELLI, M., AND CLUA, E. 2009. grmobile: A framework for touch and accelerometer gesture recognition for mobile games. In *Proceedings of the 2009 VIII Brazilian Symposium on Games and Digital Entertainment*, IEEE Computer Society, Washington, DC, USA, SBGAMES '09, 141–150.

JOSELLI, M., ZAMITH, M., VALENTE, L., CLUA, E. W. G., MONTENEGRO, A., CONCI, A., AND FEIJÓ, PAGLIOSA, P. 2008. An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu. *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 115–120.

JOSELLI, M., PASSOS, E. B., ZAMITH, M., CLUA, E., MONTENEGRO, A., AND FEIJO, B. 2009. A neighborhood grid data structure for massive 3d crowd simulation on gpu. *Games and Digital Entertainment, Brazilian Symposium on 0*, 121–131.

JOSELLI, M., ZAMITH, M., CLUA, E., LEAL-TOLEDO, R., MONTENEGRO, A., VALENTE, L., FEIJO, B., AND PAGLIOSA, P. 2010. An architeture with automatic load balancing for real-time simulation and visualization systems. *JCIS - Journal of Computational Interdisciplinary Sciences*, 207–224.

JOSELLI, M., SILVA JUNIOR, J. R., ZAMITH, M., SOLURI, E., MENDONCA, E., PELEGRINO, M., AND CLUA, E. W. G. 2012. An architecture for game interaction using mobile. In *Games Innovation Conference (IGIC), 2012 IEEE International*, 73–77.

JOSELLI, M., SILVA JUNIOR, J. R., ZAMITH, M., SOLURI, E., MENDONCA, E., PELEGRINO, M., AND CLUA, E. W. G. 2012. Techniques for designing gpgpu games. In *Games Innovation Conference (IGIC), 2012 IEEE International*, 78–82.

KADLEC, B., TUFO, H., AND DORN, G. 2009. Knowledge-assisted visualization and segmentation of geologic features using implicit surfaces. *IEEE Computer Graphics and Applications 99*, PrePrints.

KIM, T.-Y., KIM, J., AND HUR, H. 2007. A unified shader based on the opengl es 2.0 for 3d mobile game development. In *Proceedings of the 2nd international conference on Technologies for e-learning and digital entertainment*, Springer-Verlag, Berlin, Heidelberg, Edutainment'07, 898–903.

KOIVISTO, E. M. I. 2006. Mobile games 2010. In *CyberGames '06: Proceedings of the 2006 international conference on Game research and development*, Murdoch University, Murdoch University, Australia, Australia, CyberGames, 1–2.

LOPEZ, M., NYKÄNEN, H., HANNUKSELA, J., SILVEN, O., AND VEHVILÄINEN, M. 2011. Accelerating image recognition on mobile devices using gpgpu. In *Proceedings of SPIE*, vol. 7872, 78720R.

M., B. L., J., H., AND M., S. O. . V. 2009. Graphics hardware accelerated panorama builder for mobile phones. Proc. SPIE Multimedia on Mobile Devices 2009, vol. 7256. ISBN 9780819475060.

MUNSHI, A., GINSBURG, D., AND SHREINER, D. 2008. *OpenGL(R) ES 2.0 Programming Guide*, 1 ed. Addison-Wesley Professional.

MUSSE, S. R., AND THALMANN, D. 1997. A model of human crowd behavior: Group inter-relationship and collision detection analysis. In *Workshop Computer Animation and Simulation of Eurographics*, Eurographics, 39–52.

MUYAN-OZCELIK, P., OWENS, J. D., XIA, J., AND SAMANT, S. S. 2008. Fast deformable registration on the gpu: A cuda implementation of demons. In *the 1st technical session on UnConventional High Performance Computing (UCHPC) in conjunction with the 6th International Conference on Computational Science and Its Applications (ICCSA)*, IEEE Computer Society, Los Alamitos, California, M. Gavrilova, O. Gervasi, A. Lagan, Y. Mun, and A. Iglesias, Eds., ICCSA 2008, 223–233.

NAH, J.-H., KANG, Y.-S., LEE, K.-J., LEE, S.-J., HAN, T.-D., AND YANG, S.-B. 2010. Mobirt: an implementation of opengl es-based cpu-gpu hybrid ray tracer for mobile devices. In *ACM SIGGRAPH ASIA 2010 Sketches*, ACM, New York, NY, USA, SA '10, 50:1–50:2.

NORTH, R. G., 2008. Grand theft auto iv, rockstar games.

NVIDIA, 2008. Skinned instancing. Avalible at: `http://developer.download.nvidia.com/SDK/10/direct3d/Source/SkinnedInstancing/doc/SkinnedInstancingWhitePaper.pdf`.

NVIDIA, 2009. Nvidia cuda compute unified device architecture documentation version 2.2. Avalible at: `http://developer.nvidia.com/object/cuda.html`.

ONLIVE, 2012. http://www.onlive.com/.

OSTRANDER, J. 2012. *Android Ui Fundamentals: Develop & Design*. Pearson Education.

OWENS, J. D., LEUBKE, D., GOVINDARAJU, N., HARRIS, M., KRGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26(1)*, 80–113.

PASSOS, E., JOSELLI, M., ZAMITH, M., ROCHA, J., MONTENEGRO, A., CLUA, E., CONCI, A., AND FEIJÓ, B. 2008. Supermassive crowd simulation on gpu based on emergent behavior. In *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, SBC, 81–86.

PASSOS, E. B., JOSELLI, M., ZAMITH, M., CLUA, E. W. G., MONTENEGRO, A., CONCI, A., AND FEIJO, B. 2010. A bidimensional data structure and spatial optimization for supermassive crowd simulation on gpu. *Comput. Entertain. 7*, 4 (Jan.), 60:1–60:15.

PELECHANO, N., ALLBECK, J. M., AND BADLER, N. I. 2007. Controlling individual agents in high-density crowd simulation. In *SCA 07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SCA, 99–108.

PULLI, K., BAKSHEEV, A., KORNYAKOV, K., AND ERUHIMOV, V. 2012. Real-time computer vision with opencv. *Commun. ACM 55*, 6 (June), 61–69.

QUINN, M. J., METOYER, R. A., AND HUNTER-ZAWORSKI, K. 2003. Parallel implementation of the social forces model. In *Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics*, PED, 63–74.

REYNOLDS, C. W. 1987. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 25–34.

REYNOLDS, C. 1999. Steering behaviors for autonomous characters. In *Game Developers Conference 1999*, GDC.

REYNOLDS, C. 2000. Interaction with groups of autonomous characters. In *Game Developers Conference 2000*, GDC.

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

RICARDO DA SILVA JUNIOR, J., GONZALEZ CLUA, E. W., MONTENEGRO, A., LAGE, M., DREUX, M. D. A., JOSELLI, M., PAGLIOSA, P. A., AND KURYLA, C. L. 2012. A heterogeneous system based on gpu and multi-core cpu for real-time fluid and rigid body simulation. *International Journal of Computational Fluid Dynamics 26*, 3, 193–204.

RODRIGUES, R. A., DE LIMA BICHO, A., PARAVISI, M., JUNG, C. R., MAGALHAES, L. P., AND MUSSE, S. R. 2010. An interactive model for steering behaviors of groups of characters. *Appl. Artif. Intell. 24*, 6 (July), 594–616.

SARKAR, P. 2000. A brief history of cellular automata. *ACM Comput. Surv. 32*, 1, 80–107.

SHAO, W., AND TERZOPOULOS, D. 2005. Autonomous pedestrians. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, New York, NY, USA, SCA, 19–28.

SILVA, A. R., LAGES, W. S., AND CHAIMOWICZ, L. 2008. Improving boids algorithm in gpu using estimated self occlusion. In *Proceedings of SBGames'08 - VII Brazilian Symposium on Computer Games and Digital Entertainment*, Sociedade Brasileira de Computação, SBC, SBC, 41–46.

SINGHAL, N., PARK, I. K., AND CHO, S. 2010. Implementation and optimization of image processing algorithms on handheld gpu. In *ICIP*, IEEE, 4481–4484.

SINGHAL, N., PARK, I. K., AND CHO, S. 2010. Implementation and optimization of image processing algorithms on handheld gpu. In *ICIP*, 4481–4484.

SINGHAL, N., YOO, J. W., CHOI, H. Y., AND PARK, I. K. 2011. Design and optimization of image processing algorithms on mobile gpu. In *SIGGRAPH Posters*, 21.

STEED, A., AND ABOU-HAIDAR, R. 2003. Partitioning crowded virtual environments. In *VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology*, ACM, New York, NY, USA, VRST, 7–14.

THERDSTEERASUKDI, K., BYUN, G., CONG, J., CHANG, M. F., AND REINMAN, G. 2012. Utilizing rf-i and intelligent scheduling for better throughput/watt in a mobile gpu memory system. *ACM Trans. Archit. Code Optim. 8*, 4 (Jan.), 51:1–51:19.

TREUILLE, A., COOPER, S., AND POPOVIĆ, Z. 2006. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, ACM, New York, NY, USA, SIGGRAPH, 1160–1168.

TUNACODE, 2010. Cuvilib: Cuda vision and imaging library. Avalible at: `http://www.cuvilib.com/`.

VAN DEN BERG, J., PATIL, S., SEWALL, J., MANOCHA, D., AND LIN, M. 2008. Interactive navigation of multiple agents in crowded environments. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 139–147.

ZAMITH, M., VALENTE, L., JOSELLI, M., CLUA, E., TOLEDO, R., MONTENEGRO, A., AND FEIJ, B. 2011. Digital games based on cloud computing. In *SBGames 2011 - X Simpsio Brasileiro de Jogos para Computador e Entretenimento Digital*.