INTERNATIONAL JOURNAL OF ADVANCE RESEARCH AND DEVELOPMENT

(Volume 3, Issue 10)

Available online at: www.ijarnd.com

Uniform spatial subdivision to improve Boids Algorithm in a gaming environment

Prudence M Mavhemwa, Ignatius Nyangani

Student, Bindura University of Science Education, Bindura, Zimbabwe

ABSTRACT

Video games often make use of simulation to represent part of real world phenomenon; be it simulating a typical crowd behavior (e.g. chaos, rioting), or particle simulation (e.g. fire, smoke) and many other uses. Games have one common characteristic, i.e. they are interactive real-time systems, meaning to say processes that run in these applications must execute within a limited time threshold for the application to be called successful. The Boids algorithm is often used in these applications for realistic simulation of flocking type of behavior of virtual crowds. However, simulation of crowds in real-time using the algorithm is computationally time consuming, due to how the algorithm evaluates the whole crowd when searching for possible nearest neighbors for each agent in the simulation. There are several approaches to improve performance of these flocking simulations in real-time, and in this document we discuss some of those methods that have been applied to the Boids algorithm. We further implement and test one of these performance optimization methods, and use benchmarking results to compare performance of the method versus the boids algorithms' brute force neighborhood gathering approach.

Keywords— *CPU, GPU, GPGPU, Interactive systems, Agent, Crowd, Simulation model, 2D, 3D, Crowd simulation, AI*

1. INTRODUCTION

Several algorithms have been developed to simulate the coordinated motion of a group of agents, and most of them have originated from swarm intelligence. They are most known as flocking algorithms, and have been widely implemented in video games to simulate the complex flocking behavior of interacting agents such as soldiers and monsters in virtual environments.

The first flocking-behavior simulation was done on a computer by Craig W. Reynolds in 1986, and called his simulation program "Birds" (M.Sajwan, 2014). He coined the name 'boid' to represent each agent in his simulation, hence the name of the algorithm. The algorithm tries to simulate complex flocking behavior by implementing three main heuristic rules that each boid must obey which are:

- **Cohesion**: Cohere with other characters by steering towards the average position of the local neighborhood.
- **Alignment**: Steer towards the average direction or heading of local flock mates.
- **Separation**: Avoid collision and crowding with other local flock mates.

When combined, these three basic rules can provide a realistic flocking simulation of a group of agents. Unfortunately, the Boids algorithm can be very computer-intensive when very large groups are considered, mainly due to the process of evaluating possible nearest neighbors (Reynolds, 1987). By using the algorithm in real-time applications such as video games, the cost of simulating large crowds can significantly affect application's performance.

1.1 Problem Statement

Real-time simulation of virtual crowds has always been a big challenge, mainly due to the limited amount of computational resources available on commercial CPUs (Yilmaz, 2010). In the boids simulation model, for each boid to make a decision such as to move in space, it must first determine the possible nearest neighbors that it might (for instance) collide with. The problem is how one boid determines its proximate neighbors out of a big population of boids, whilst using limited computational resources that are inherent in real-time systems. The brute force approach of determining nearest neighbors makes it infeasible for the boids algorithm to simulate large crowds in real-time on most consumer CPUs. In real-time interactive systems, time is a limited resource and processes must execute within some given time threshold in-order for the system to be called successful (Joselli, Passos, Silva Junior, Zamith, & Clua, 2012). Thus, some performance optimization methods have to be explored in order to minimize the total simulation time so as to meet system's time constraints.

1.2 Objective

To determine if implementation of uniform spatial subdivision to the boids algorithm may speed up flocking simulation of large crowd sizes in real-time.

2. LITERATURE REVIEW

The boids algorithm is one of the most commonly used algorithms in simulating crowd behavior, and has been used as a back-borne for most of the algorithms used in crowd simulation (Sajwan, Gosain, & Surani, 2014). However, naïve implementation of the boids algorithm has been shown to have a complexity of $O(n^2)$ as a direct result of near-neighbor queries (Silva, Lages, & Chaimowicz, 2009). This section will address how some performance optimization methods have been used to accelerate performance of the boids algorithm in simulating large virtual crowds with emphasis on the effectiveness of these strategies in real time applications.

2.1 Crowd Simulation Models

Generally, all crowd simulation models can be classified under macroscopic and microscopic models, depending on their properties and approach in simulating crowds (Afanasyeva & Afanasyeva, 2014). Macroscopic models treat the crowd as a whole and use flows to characterize crowd behavior and movement, whilst Microscopic models consider movement and behavior individually for each agent (Sun, 2014).

2.1.1 Macroscopic Models: Macroscopic models are based on observed flow of human motion in different scenarios, for instance, panic or a crowd of people evacuating a building. As such, these models are used to simulate human behavior as an aggregate flow or in groups and avoids per-agent processing(Treuille, Cooper, & Popović, 2006). Despite the performance benefits that this method offers, it lacks the flexibility of a full agent-based approach since decisions are made at global level rather than per individual.

2.1.2 Microscopic Models: In microscopic models, each agent in the simulation is regarded as an independent decision making entity. As the simulation progresses, agents continuously examine their surrounding environment, and then each agent will then make its own decisions depending on the current situation (Afanasyeva & Afanasyeva, 2014). These decisions will then be used to drive the agent's individual characteristics such as position and velocity (Fachada, Lopes, Martins, & Rosa, 2016). However, since each agent is an independent entity that can make its own decisions, this also increases the amount of computations that have to be performed per agent in each simulation run.

2.1.2.1 Boids Model: The algorithm in this model is able to simulate complex flocking behavior as a result of coordinated motion by implementing three simple rules that define the steering behavior of each boid. The following are the three rules as described earlier Separation, Alignment, and Cohesion**.**

The separation rule simulates collision avoidance from nearby flock mates by calculating a force, such that if applied to a boid's velocity, the force will allow a boid to get away from close agents. A threshold distance of separation is used to determine if the other boids in the flock are close enough to get away from.

Alignment rule allows a boid to steer towards the average velocity or heading of the flock, essentially making a boid to steer towards the nearest boids. Nearest boids are determined by a threshold alignment distance, and the average of their velocities is calculated. Then the difference between the average velocity and the current boid's velocity are measured and the force is added to the direction of the vector difference of these two velocities.

The cohesion rule allows a boid to get closer to the center of neighbor boids, and this rule can be seen as a type of attraction rule. The center of neighbors is calculated by adding all their positions and then dividing them by the number of the neighbors.

A generalized pseudo code for the naïve boids algorithm is presented below (Weiss Robin M, 2010)

create N boid agents and initialize with random position and velocity
for each boid:
    vAvoid = collision avoidance force
    vMatching = velocity matching force
    vCentering = flock centering force
    *boid.velocity += (c1 * vAvoid) + (c2 * vMatching) + (c3 * vCentering)*
    boid.position += boid.velocity
end for

The terms *c1, c2* and *c3* represent the weights used to weigh the forces, depending on the type of agent being simulated. For instance, when simulating the schooling behavior of fish, a high weight on the cohesion force will be used. Humans however, might need a small weight of the cohesion force since human movement tend to be more independent in nature, unless maybe simulating rioting scenarios where people seem to move cohesively in groups.

Other than these basic rules, several other rules can be added to enhance the visual results of the simulation (C. W. Reynolds, n.d.-a). The advantage of the boids algorithm over most of contemporary simulation algorithms is its ease of implementation whilst providing highly realistic results (Afanasyeva & Afanasyeva, 2014). Since it is a microscopic model, these three forces are calculated independently of other agents in a simulation. This individuality allows simulation of more complex behavior since each agent reacts to its surrounding environment basing on its current situation.

Nevertheless, the boids algorithm has its own shortcomings. The straightforward implementation of the boids algorithm has an asymptotic complexity of $O(n^2)$ (Joselli et al., 2012). For small flock a size, the computational cost is not of much concern but when the size of the flock starts to increase, visualization of the simulation in real-time becomes very expensive. Thus, applying the algorithm to simulate crowds in real-time applications becomes useless.

2.2 Boids Algorithm Performance Optimization

The major bottleneck of the naïve boids algorithm is the cost of performing near-neighbor queries in-order to gather separation, cohesion and alignment information (C. Reynolds, 1983), (Silva et al., 2009). For each agent in the boids simulation to determine its proximate neighbors, it has to compare its distance from every other agent's position in the scene in-order to determine the number of agents that are within its neighborhood (Yilmaz, 2010). For very small population sizes, the computational time is not of much concern. However, as the population size starts to increase to several hundreds or thousands of agents, these proximity costs will ultimately dominate all other costs in the simulation, regardless of how fast each proximity query can be done (Reynolds Craig W, 2000). This makes the naïve approach useless in real-time interactive environments.

2.2.1 Parallel Processing: Parallel processing is a field of computation where several processors are used to solve parts of a given problem in parallel. Parallelism of a model can be achieved by decomposing the model into several components, such that each component can be independently processed by logical processors in a parallel manner (Fachada et al., 2016).

Several researchers have used parallel processing to speed up the boids algorithm in simulating massive crowd sizes, both on the CPU and the GPU. The former approach uses multi-threading as a way of performing these computations on different processor cores, whilst the latter uses either GPGPU or modern heterogeneous systems such as NVidia's Compute Unit Device Architecture (CUDA) or Open Computing Language (OpenCL) (Yilmaz, 2010), (Drozd, 2015), (Fachada et al., 2016).

The boids algorithm was also used to simulate swarming behavior by (Drozd, 2015), in which agents used information from their surrounding environment to perform given tasks such as food foraging. He showed that by exploiting multicore CPUs for parallel processing, he could simulate signal propagation for over 100000 agents at interactive frame rates.

The computational power of GPUs and their ability to run non graphical algorithms through GPGPU has motivated other researchers to exploit parallel programming to speed up crowd simulation using the boids algorithm. Modern GPUs have been shown to run algorithms 10 to 100 times faster than CPUs (Relations, n.d.). (Yilmaz, 2010) used CUDA technology as the general purpose parallel programming tool to write the parallel boids algorithm for the NVidia GTX 295 GPU. He showed that it is possible to achieve two orders of magnitude speedups using CUDA parallel computing architecture as compared to serial code running on the CPU.

Another usage of parallel processing in the boids algorithm was implemented by (Zhou, 2004). His main goal was to minimize the quadratic complexity and to increase the number of agents in the simulation. In their implementation, they used a computer cluster and each processor had to communicate with other nearby processors to query for near-neighbor information. Using this approach, they were able to simulate up to 512 individual boids at interactive frame rates.

Despite the performance improvements that parallelism offers to the boids algorithm, it also presents several problems; mainly, the effective way in which the model must be represented in order for parallelism to work (Fachada et al., 2016). Parallel processing is a divide and conquer technique, in which a problem in decomposed into several sub problems that can be solved in parallel. Therefore in order to use parallel processing on a model, there is need to decompose the model into several components that can be processed independently. Fachada argued that model decomposition can involuntarily introduce changes which can modify the model dynamic due to implementation details, thus failing to reproduce same visual results or behavior as the original model.

2.2.2 Occlusion Culling: (Silva et al., 2009) introduced an approach to minimize the costs of near neighbor queries in the boids algorithm using estimated self-occlusion. He defined self-occlusion as 'the possible occlusion caused by a boid in view of the other boids during flight'. The main idea was that boids must only query near neighbor information from other boids that are visible from their field of view, so as to perform separation, alignment and cohesion calculations. It is shown in Figure 1 below.

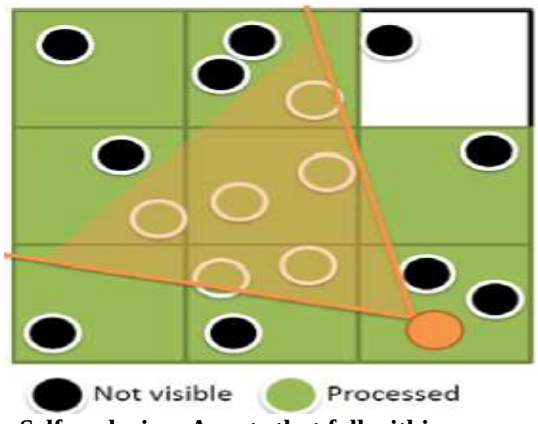

Fig. 1: Self-occlusion. Agents that fall within an agent field of view and un-occluded by other agents are considered as neighbors (Silva et al., 2009)

Using this approach, the number of neighbors considered in the computations is significantly reduced as a result of neglecting boids that would be invisible due to the presence of other boids. The implementation of the boids model and the visibility culling algorithm was done on the GPU using both GPGPU and CUDA. Despite being efficient at simulating large flock sizes, the method relies on offloading visibility computations to the GPU for performance speed-ups. This limits the applicability of this technique in some computers that have CPUs only as a computational device.

2.2.3 Spatial Subdivision

Spatial subdivision can be defined as the structured partitioning of geometry (Rhodes, 2014). It n is a general optimization technique that has been applied to several computational problems including ray tracing (Havran, 2004) and cloth simulation (Ho, Geoff, & Fabio, 2012). The basic idea behind spatial subdivision is to subdivide a given geometrical space into smaller sub-spaces or cells, which can then be used to speed up proximity or locality queries (C. Reynolds, 2006).Figure 2 below shows the Spatial Subdivision technique.

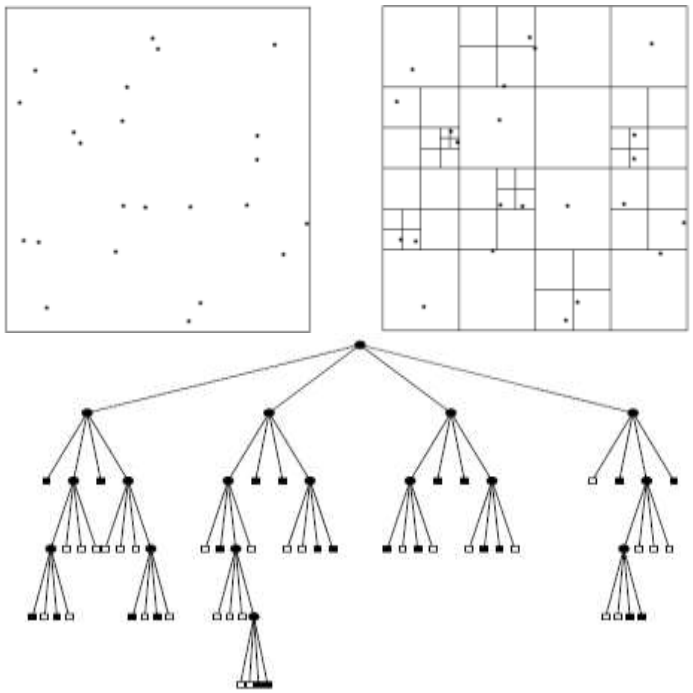

Fig. 2: **Illustration of 2D space decomposition by using a quadtree. Points in 2D space (top left), subdivided 2D space (top right), resulting quadtree structure (bottom) (Drozd, 2015)**

Generally, spatial subdivision structures can be categorized into two sub categories; object independent structures and object dependent structures (Teschner, Detection, & Response, 1976).

2.2.3.1 Object independent spatial structures: Object independent spatial structures do not take into account the distribution of the objects in space. These structures divide a given space independent of object distribution, and are considered to be the simplest spatial structures. Examples of object independent structures are regular grids and bin-lattice method.

2.2.3.2 Bin-Lattice Spatial Subdivision: In this method, a region of space is subdivided into smaller boxes called bins (C. Reynolds, 2006). At the beginning of the simulation, agents are distributed into these bins depending on their initial position. Every time an agent moves from its current position to the next

position, it checks if it is still inside its allocated bin. If the agent has moved to another bin, it updates its bin membership to the current new bin. All the bins that intersect with the agent's spherical neighborhood become the current search space, as such, near neighbor queries will be limited to those bins.

Using this type of optimization structure, Reynolds was able to reduce the simulation complexity from O(n^2) to linear O (kn). This was largely due to a constant bounded number of agents k within each boids' radius of influence that were considered as near neighbors, thus, avoiding a boid to evaluate the whole population. However, this complexity has been shown to be highly dependent on the maximum density of each grid cell, which could be high if the simulated environment is very big and condensed (Joselli et al., 2009). Reynolds showed that by using bin-lattice subdivision technique on a flock size of 1000 simulated birds, performing locality queries was about 16 times faster than the naïve implementation.

2.2.3.2 Uniform Spatial Subdivision: The main idea behind uniform spatial subdivision is to partition the space into many uniform cells, and then perform tests only for primitive pairs belonging to the same cell (Ho et al., 2012). The resulting data structure is often referred to as a uniform or regular grid. As the space is divided into smaller chunks and computation are limited to a single cell only, selecting the possible near neighbors will avoid the O(n^2) complexity (Silva et al., 2009).

The major advantage of using a uniform grid is that the grid doesn't change over the simulation process, thus, it has a constant cost to build as compared to other recursive structures such as quadtrees (Silva et al., 2009). Despite recursive structures being more efficient in most scenarios, uniform grids are much simpler and less complex to construct and manage. However, regularity of the grid does not allow the structure to adapt with distribution of objects in a scene (Havran, 2004).

2.2.3.3 Object dependent spatial structures: Object dependent spatial structures consider the distribution of objects in space, which result in irregular subdivision. These type of structures ,also known as hierarchical structures, partition a given space into several subspaces, taking into account the distribution of objects in a scene (Hughes et al., 2013). By taking into account object distribution in a given space, hierarchical structure results in a non-uniform subdivided space; areas where objects are densely populated is subdivided the most as compared to other areas with fewer objects. Under these we have Quadtrees(Lu,2014), (Bennet 2013) and (Devlin,2016) and Octrees(Samet ,1988) , (Pantazopoulos & Tzafestas, 2002) and (Hughex et-al ,2013).

Using this data structure has the advantage of eliminating large portions of the flock from near neighbor consideration (pruning). Thus when used for near neighbor queries, the structure may speed up computations by orders of magnitude from the naïve approach. However, Devlin's implementation was inefficient because the quadtree had to be recreated in each simulation run. He proposed a solution to this problem in which a tree would be saved but this approach can be complex and expensive to implement in real-time, by considering the time cost of building the structure with respect to the crowd size that is represents.

Octrees have been used to speed up a flocking boids simulation by (Drozd, 2015). His implementation was based on signal-driven evolving autonomous agents using an asynchronous evolutionary algorithm. The algorithm was based on signal propagation, and an aggregate signal transmitted by all other boids in the simulation is calculated by each boid, which can lead to an O(n^2) complexity if naïve algorithm was used. Similar to (Devlin, 2016), each simulation step began by constructing the tree from scratch, and performing near neighbor queries for each agent by traversing up the constructed tree. He noted that his algorithm was a few orders of magnitude faster than naive implementation. Regardless of the octree improving performance of the flocking simulation, octrees are just 3D versions of quadtrees and suffer from the same limitations, such as tree balancing and frequent re-construction. Each simulation step required that the tree be reconstructed, which may be inefficient for very large crowds.

2.3 Challenges of using uniform grids

Uniform grids has a lot of advantages when used as an acceleration structure; mainly its constant cost to build and simplicity of implementation(Devlin, 2016) and (Drozd, 2015). However, the manner in which objects are distributed over the grid is a major determination of success of this structure (Rhodes, 2014) . The worst case result when all agents in the population are crammed within a single cell on the grid. This means that the whole population size (n) will be evaluated by each boid in that cell during near-neighbor querying, thus leading to the same evaluation as the O(n^2) brute force approach. Possible solutions to this problem have been proposed by researchers; notably (Havran, 2004) proposed to recursively subdivide the cell that has most objects, so that it can be further broken down into smaller cells. However, considering the fact that boids are dynamic and the whole population might move to another single cell, this means that the recursive subdivision process must also be done on that cell to break it into smaller cells. As a result, there would be frequent subdivision which may result in a dense grid which may be difficult to manage memory wise.

2.4 Conclusion

The researchers have presented and evaluated some of the performance optimization techniques that have been applied to the boids algorithm. The researchers are going to make use of a uniform grid in order to speed the boids algorithm in simulating a virtual crowd. However, performance of uniform grids have been shown to be affected by irregular allocation of objects across its cells, which may lead to the same evaluation as the naïve algorithm. To counter this limitation, the researchers are going to use a goal driven boids simulation, in which a boid is driven by a set of goals that avoid it from moving into cells that are already densely allocated. This will greatly reduce the possibility of occurrence of a worst case scenario, whilst maintaining the structure of the grid as compared to the recursive cell-subdivision solution.

3. RESEARCH DESIGN AND METHODOLOGY

The researchers used a mixed methodology which includes the modeling of the uniform grid algorithm, simulation of a crowd using the algorithm and an experiment to test the algorithm using a set of parameters. The simulation parameter that was varied during the experiment was the input size (i.e. crowd size), and the researcher aimed to describe the variation of the algorithm output under the condition that was hypothesized to reflect variation.

3.1 Crowd simulation design framework

The crowd was represented as a collection of individual humanoid agents (boids) that can interact on their own, and also to the user's (game player) inputs. Uniform spatial subdivision

concept was used to decompose the game world into smaller regular cells, which were subsequently used as input to the boids algorithm in order to speed up near-neighbor queries during the simulation.

The game world was represented as a rectangular 3D mesh covering every object in a modeled game environment. The subdivision process was done using a 3D authoring tool, and the resulting grid of uniformly-subdivided cells was stored in an array as a collection of small bounding boxes of those cells. Thus, crowd simulation design was comprised of two parts; the preprocessing of scene to create a uniform grid and the actual utilization of that grid as an optimization structure during the simulation.

3.1.1 Scene pre-processing: The subdivision process was not done algorithmically, but was done manually through modeling in a 3D authoring tool called Blender. This was done during the scene modelling stage when the actual game objects were created. The output of the modeling process (uniform grid mesh) was exported into a game engine, and then used to extract geometric information that was used for the construction of a uniform grid of 3D axis-aligned bounding boxes. Lastly, simulation was carried out by allocating the boids to cells on the grid basing on their position in space, and allocating them random goals.

3.1.2 Creation of Uniform Grid: In Blender, the world object is represented as a huge cubic 3D mesh covering all other objects in the scene as shown in Figure 3 below.

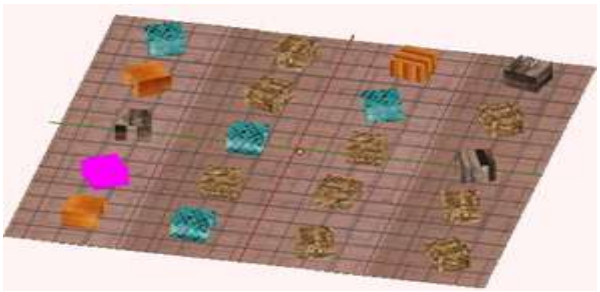

Fig. 3: A prototype game environment scene with buildings (snapshot taken from blender)

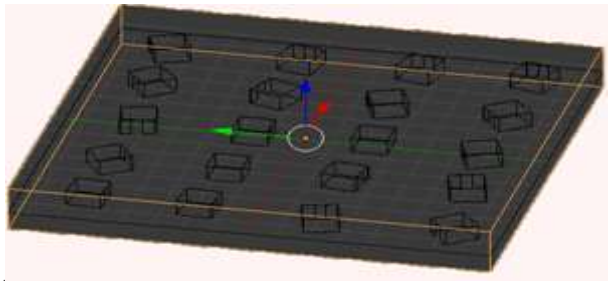

Fig. 4: The yellow outlined box shows the rectangular world mesh covering all objects in the game scene (snapshot taken from blender)

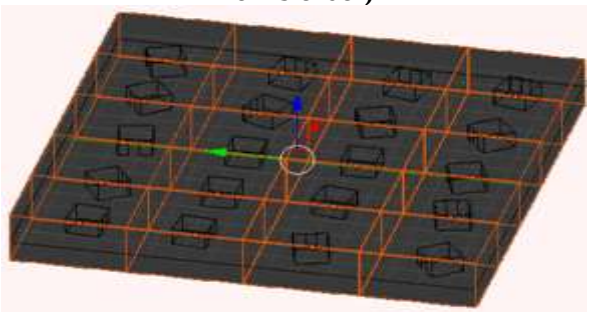

Fig. 5: A subdivided world mesh giving a regular grid of cell meshes over the scene (snapshot taken from blender)

Uniform subdivision was done by editing the 3D world mesh using Blender's mesh editing tools, specifically the loop-cut tool which is often used to add uniformly spaced edges on a mesh. After the loops are added to the world mesh, the small cells marked by the added ridges were then split into independent objects from the main world mesh. Each cell mesh was then centered about its own geometry in the scene, and shared the same identifier as the main world object, as shown above. These subdivided meshes were used as a guide for representing the uniform grid in the game engine.

After the scene was modelled and saved, it was then exported into the Irrlicht game engine where the actual 3D bounding boxes were extracted from the subdivided cubic meshes' geometry. In Irrlicht, each mesh is imported as a scene node and added to the collection of other scene nodes called the scene graph, and each scene node has its own bounding box automatically generated by the engine when the node is created. All the nodes that represent a cell in the grid are given a same identifier to distinguish them from all other nodes in the scene graph, and to identify them for pre-processing.

The extracted bounding boxes were subsequently stored in an array and the cubic meshes were removed from the scene graph. Thus, the uniform grid was represented as an array of 3D axis-aligned bounding boxes extracted from the cell meshes, not the actual mesh themselves. This pre-processing step was done once before the simulation process, and the grid size remains constant throughout the simulation process

3.1.3 Cell and goals allocation: The distribution of objects using a uniform grid has a severe impact on the grid's performance, and to achieve optimal performance, each cell in the grid must contain the same number of objects as all other cells (Rhodes,2014). To avoid irregular allocation of boids over the uniform grid, a cell and goal allocation algorithm was used to equally distribute the crowd to the cells and allocate them goals in those cells. The maximum capacity that each cell can hold was achieved by dividing the crowd size with the number of cells in the grid. The cell allocation algorithm was subsequently used to fill each cell up to maximum capacity and move on to the next cell recursively. Maximum capacity is calculated as follows:

$$maxCapacity = crowdSize/GridSize$$

The goals are just a set of 3D positions that were located inside the boid's allocated cell. These positions help the boid to randomly navigate throughout the virtual environment. Goals are extracted from *'empty'* scene objects in the game scene that were designated as goal nodes during the scene modelling process; they just contain 3d position information, a name and an identity to uniquely identify them.

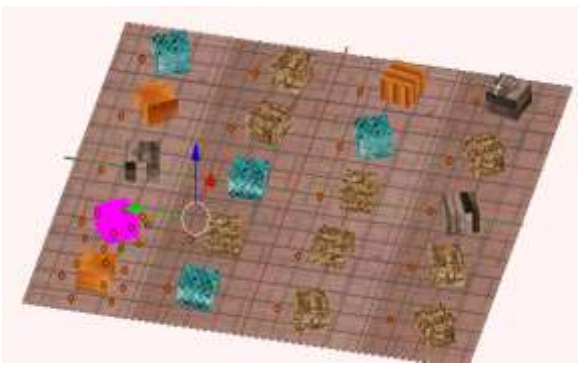

Fig. 6: The orange dots represents the 'empty' objects that are used to represent the goals of the agents

A boid continually steer towards its goal taking into consideration the separation, alignment and cohesion forces. If a goal is reached, another goal is randomly chosen in that cell and set as the current goal. Goal setting simulates random movement as usually seen in human crowds (Szymanezyk, Duckett, & Dickinson, 2012).

3.2 Boids Algorithm and Utilization of the Uniform Grid an Optimization Structure

Since the major goal of this research is to determine if the implementation of spatial subdivision can speed up crowd simulation using the boids algorithm, the researchers implemented two different algorithms; the naïve neighborhood gathering approach of the original boids algorithm and a modified algorithm which used the uniform grid as neighborhood gathering structure. The runtimes of these algorithms were then measured and compared against each other to determine the most effective approach. The boids algorithm was adapted from (Shiffman, 2016).

3.2.1 The naïve implementation: The naïve approach considers the whole population as the possible neighbors, and calculates a separation force which is then used to steer the agent. The pseudo code as implemented in the game application is given below:

```
Procedure: BoidsAlgorithm(BoidList)
    for each boid in BoidList
        if boid != alive
            continue
        end if
        if boid.identity == thisBoid.identity
            continue
        end if
        distance = getDistanceBetweenPoints(boid.position,thisBoid.position)
        if distance<seperationdistance
            dir = thisBoid.position – boid.position
            dir =dir/distance
            seperationforce = seperationforce+dir
            seperationcount =seperationcount+1
        end if
        if distance<cohesiondistance
            cohesionforce = cohesionforce+boid.getPosition
            cohesioncount = cohesioncount+1
        end if
        if distance<alignmentdistance
            alignmentforce = alignmentforce+boid.getVelocity
            alignmentcount =alignmentcount+1
        end if
    normalize seperationforce,alignmentforce and cohesionforce
    thisBoid.setVelocity(thisBoid.getVelocity + seperatiforce+alignmentforce+cohesionforce
end for
```

3.2.2 The Uniform Grid implementation: The uniform grid approach was comprised of two stages. The first stage is the allocation of a *cell index* which shows the cell that the boid is currently inside. The second stage performs continual update of the boid's *cell index* in case the boid moved away from its previously allocated cell, and the calculation of the separation, cohesion and alignment steering forces by making use of the *cell index* to retrieve possible nearest neighbors in the cell that it references. The Pseudocode is outlined below:

```
Procedure:
boidsAlgorithmUgrid(Grid,thisBoid,sepDist,velocity,cellindex,isRegistered)
start
if cellindex==infinity
    for each cell in Grid
        if thisBoid.position is inside cell
            isregistered = registerBoidInCell(cell,thisBoid)
            cellindex = cell.getIndex()
            goto start
        end if
    end if
end for
end if
    currentcell = Grid.getCellByIndex(cellindex)
    isboidstillinside = currentcell.isPointInside(thisBoid.position)
        if isboidstillinside
            CellBoidList = currentcell.getList
            BoidsAlgorithm(CellBoidList)
        else

    currentcell.removeBoidFromCellList(thisBoid)
            cellindex =infinity
        end if-else
end if
```

The '*isregistered'* flag can only be true if the size of boids registered in that cell is less than *maxCapacity*. If this flag was set to false, the boid will quit its current desire goal of moving to the area enclosed by that cell, thereby reducing the possibility worst case scenario of all boids accumulating in a single cell.

3.2.3 How goal allocation was used to avoid performance drawback: When a boid moved into a region of space (cell) that was allocated up to *maxCapacity* value, its '*isregistered*' flag was set to false. The AI of the boids was designed in such a way that, if a boid is unregistered, it discards its current desired goals and choose a random goal that fall in its default allocated cell at the beginning of the simulation. This essentially resets the boid to its default cell by pursuing that goal, minimizing the possibility of boids packing up in a single cell which can greatly affect performance.

3.3 Game Application Design Framework

The crowd simulation framework was implemented in a first person shooter video game called HERO. In the game, a player controls a character that is tasked with a main goal of rescuing a hostage at a given checkpoint in a game world. The hostage is being kept by the 'enemies', and the player's goal is to find and rescue the hostage without being detected by the enemies. If detected, the enemies will flock towards the player and attack him. The 'enemies' represents the crowd of boids in the game environment. Other than visual detection, other actions such as player's actions such as gunfire can alert nearby boids to search the area where the gun was fired, and if player is found, the boids will flock towards and attack the player.

3.3.1 Boids Navigation: The games virtual environment was designed to mimic a city with a lot of objects such as buildings and walls. Each boid must avoid collision with these objects when moving along its desired goal, including other boids in the simulation. Due to the complexity of the virtual scene, a path planning algorithm was used for calculating obstacle-free paths.

The researchers used greedy best-first search algorithm for path planning. As indicated by (Afanasyeva & Afanasyeva, 2014), a simplified representation of the game environment (a graph) was used by the path planning algorithm to calculate obstacle-free paths for the boids in the simulation. Given a *start* node and the *goal* node, the algorithm approximated a collision-free path by expanding nodes on the graph that are nearest to the goal node. The algorithm guarantees a collision free path only with static obstacles in the environment.

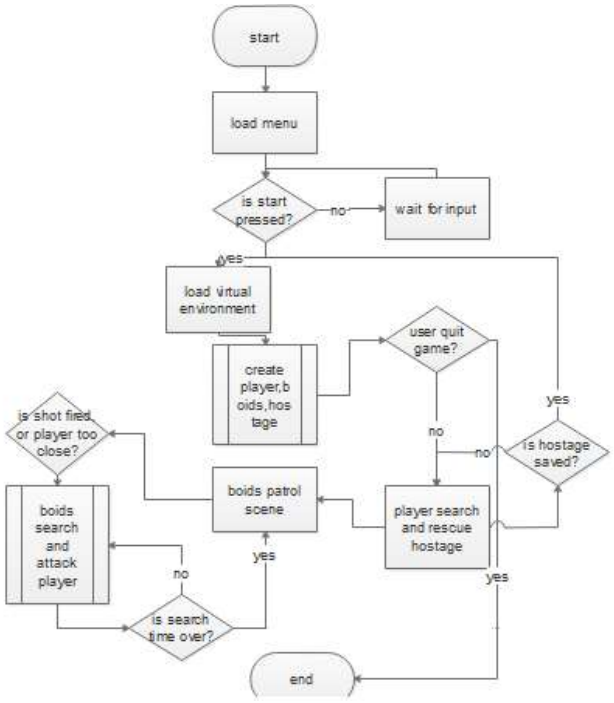

Fig. 7: Flowchart showing application functionality

3.3.2 Boid Modeling: The 3D models representing the boid agent in the game were designed to represent human models. The virtual human models were defined by their mesh, their skeletons, and their set of textures. Different elements were varied across models such as texture color as indicated by (Thalmann, Grillon, Maim, & Yersin, 2009). Modelling was done using Blender software.

3.3.3 Boids Animation: The 3D models were pre-animated using motion capture data from Carnegie Mellon University's motion capture database files. The motion capture data from the files was mapped onto the skeleton of the 3D mesh that represents the boid so that the boid may replicate actual human motion captured on camera. BVHacker software was used for editing motion capture data, and Blender was used to map that data onto a models' skeleton.

3.4 Sampling

Random sampling was used for determining the input size of the algorithm, i.e. crowd size, and the samples are captured uniformly during the simulation run. Each sample was captured after every five simulation runs, and two variables were captured; the profiled average run time of the simulation algorithm and the average time of simulating a single boid. The following table outlines a sample frame structure.

Table 1: **Structure of a sample frame**

Frame number	Average time/population	Average time/single boid

3.4.1 Profiling execution times: To calculate average times for each sample frame, the total execution for each algorithm time was measured by using high resolution timestamps using Windows API method, *QueryPerfomanceCounter()*. This function 'retrieves the current value of the performance counter, which is a high resolution (<1μs) time stamp that can be used for time-interval measurements' (Microsoft, 2017). These counters can provide information as to how well an algorithm is performing.

The usage of the high resolution time stamping method was chosen as it is typically the best method to use to time stamp events and measure small time intervals that occur on a system. The only limitation is that it is a Windows API and only work on Windows, thus, experiments will only be limited to the Windows operating system.

3.5 Data Analysis Procedure

The researcher used quantitative analysis on the results obtained from the experiments. The average execution time values (means) obtained by executing the algorithm under same conditions were afterward used to conduct a chi-squared test to determine if there exist a significant difference in performance after the uniform spatial subdivision was implemented to the naïve algorithm. The tests were conducted using GNU's PSPP statistical package. This analysis procedure helped to verifying the research hypothesis.

4. DATA PRESENTATION, ANALYSIS AND INTERPRETATION

4.1 Data presentation

Profiling was used to gather data for the experimental measurement of the performance of the algorithms using time stamping, and a benchmark was conducted to assess the relative performance of uniform grid to the naïve implementation. An experiment was carried out for each algorithm, and each experiment is intended to execute that algorithm on an input problem of size *n*, that is, the crowd size.

Samples were taken after every five simulation runs, and two variables were captured in each sample; the average time taken to simulate the whole population and the average time taken to simulate a single boid in the population. A total of 5000 samples were taken for each simulation run. Benchmarking was done on an HP 655 Laptop with 4GB RAM, an AMD E2-1800 APU 1.70Gz processor and 64-bit Windows 7 Ultimate operating system. Table 2 below shows summaries.

Table 2: **Simulation times of two algorithms**

Simulation run	Crowd Size	Naïve Approach		Uniform Grid Approach (3x3 grid)	
		Avg Time/Crowd(µs)	*Avg Time/ Boid (µs)*	*Avg Time/Crowd(µs)*	*Avg Time/ Boid (µs)*
1	9	50.843	5.649	39.8429	4.427
2	12	64.961	5.413	46.662	3.889
3	86	1761.944	20.488	468.072	5.443
4	386	35273.175	91.382	6991.424	18.112
5	469	49011.579	104.502	9203.373	19.623
6	590	82597.872	139.996	14948.575	25.337
7	855	212265.702698	248.263980	33798.552013	39.533951
8	967	329499.812	340.744	47419.902	49.038
9	1290	640414.366	496.519	108848.248	84.372

Average time per simulation run is calculated as:

$$\frac{Total\ Simulation\ Time}{Total\ number\ of\ Simulations\ (runs)}$$

Average time to simulate a single boid is calculated as:

$$\frac{Average\ Time\ per\ Simulation}{Ttal\ Number\ of\ Boids}$$

4.2 Data analysis and interpretation

A chi-square test was carried out on the means (average execution times) in the experiment. The aim was to determine if there is a significant difference in performance as a result of the grid implementation. Statistic results obtained in PSPP from the above experiment are given below:

Table 3: Chi-square test results

Summary.

	Cases					
	Valid		Missing		Total	
	N	Percent	N	Percent	N	Percent
naive algorithm * uniform grid implementation	9	100.0%	0	0.0%	9	100.0%

Chi-square tests.

Statistic	*Value*	*df*	*Asymp. Sig. (2-tailed)*
Pearson Chi-Square	72.00	64	.230
Likelihood Ratio	39.55	64	.993
Linear-by-Linear Association	7.95	1	.005
N of Valid Cases	9		

Figure 9 below shows the graphical representation of the comparison.

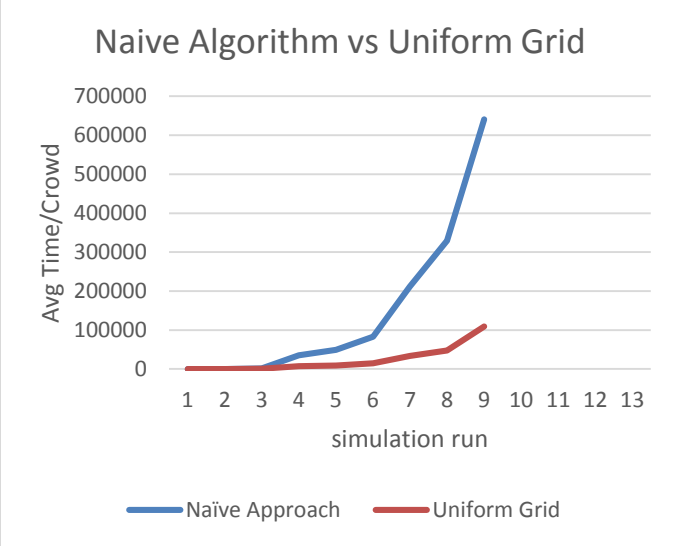

Fig. 8*:* Naive vs. Uniform grid results

According to the results of implementation of the 3x3 uniform grid, best performance speedups were achieved when crowd sizes were large as compared to smaller crowd sizes. This is largely the result of the cost of determining the current cell on the grid that each boid resides in each simulation run. For a small crowd size, the naïve approach of neighborhood gathering is much efficient because there are very few neighborhood comparisons that are made and there are no costs of cell allocation. The uniform grid, though faster than naïve approach for the same small crowd size, is less effective because the algorithm has an additional cost of determining the cell that each boid lie before determining a Boid's nearest neighbors.

However, as the crowd size is increased, the uniform grid becomes more efficient than the brute force approach because it discards a large amount of the crowd from neighborhood tests as compared to the naïve approach. Moreover, the goal allocation strategy used by the researchers limited each cell's carrying capacity to values less than the total population size, which avoided the boids to gather in a single cell, thus avoiding the same order of evaluation as the naïve approach. In other words, increasing the number of boids leads to a significant performance improvement when using the uniform grid because it largely minimizes unnecessary neighborhood tests. On the contrary, small crowd sizes do not result in best speedups because there are few neighborhood tests that need to be performed and there is are greater costs of cell allocation.

5. SUMMARY OF RESEARCH FINDINGS

To find the difference between the algorithms execution times, time stamping method was used to measure the interval between each algorithm start times up to its finishing time. Results obtained indicated that the uniform grid implementation performed better at large crowd sizes as compared to the naïve neighborhood gathering. Moreover, goal setting allowed the crowd to be uniformly distributed across the environment during the simulation, thus, minimizing the possibility of a worst case scenario whereby the whole population is cluttered in a single cell.

5.1 Conclusion

The aim of the researcher was to determine if implementation of uniform spatial subdivision in the Boids algorithm may speed up simulation of large crowds. By using the research results gathered, we can say the research objectives were achieved to a greater extent.

The statistical results obtained from chi-square test produced a p-value of 0.230. Using the test, we reject H_0 if p is less than 0.05. In this case, we would fail to reject H_0 and conclude that there is no significant difference of using the uniform spatial subdivision approach in improving performance of the naïve boids algorithm. However, by *improving* performance, we actually mean reducing the actual execution time, not increasing it. This means there exist weak evidence that there is no significant difference of using the uniform spatial subdivision approach in increasing performance of the naïve boids algorithm, thus, we reject H_0 and conclude that there is a significant difference of using the uniform spatial subdivision approach in increasing performance of the naïve boids algorithm. Thus the researchers was able to accelerate performance of the naïve Boids algorithm by using uniform spatial subdivision technique. We can safely say the implementation of uniform spatial subdivision improves performance of the Boids algorithm in simulating flocking behavior in large crowds. The benchmark results indicated that the researchers were able to accelerate performance, since we were able to simulate up to 1290 boids at interactive frame rates as compared to the naïve algorithm.

5.2 Recommendations and future work

Due to the performance results obtained in this research, we recommend implementation of the uniform grid technique when optimizing simulations that involve a lot of interacting objects in real-time, be it crowd or particle simulations. Its simplicity of implementation in combination with using a set of goals to constrain objects movement, can offer a great deal of performance while ensuring uniform distribution of objects on the grid to avoid performance hitches. We have shown that this technique was able to simulate more than 1000 boids at interactive rates. A possible combination that might be exploited in the future is to combine parallel processing methods (e.g. multithreading) with the uniform grid to further improve the grid's performance.

6. REFERENCES

[1] Afanasyeva A. (2014, June). Developing a crowd simulation libraryfor mobile games. BACHELOR´S THESIS. TURKU UNIVERSITY OF APPLIED SCIENCES.

[2] Bennet H, Y. C. (2013). Amortized Analysis of Smooth Quadtrees in All Dimensions. 1-2. New York: Courant Institute.

[3] Beyer K, G. J. (1998). When is "Nearest Neighbor" Meaningful? Springer-Verlag Berlin Heidelberg , 217-235.

[4] Daniel Thalmann, H. G. (n.d.). Challenges in Crowd Simulation. Retrieved October 2, 2016, from https://infoscience.epfl.ch

[5] Devlin C. (2016, August 22). An Investigation into an Assortment of Flocking Algorithms. Masters Thesis, 1-14.

[6] Drozd A. (2015). Signal-Driven Swarming: A Parallel Implementation of Evolved Autonomous Agents to Perform A Foraging Task Signal-Driven Swarming.A . 126-133.

[7] Fachada N, L. V. (2016). Towards a standard model for research in agent-based modeling and simulation. International Journal of Parallel Programming, 1-33.

[8] Ho T, G. W. (2012). Virtual Subdivision for GPU based collision detection of deformable objects using a uniform grid. Springer-Verlag, 829-838.

[9] Hughes J, V. D. (2013). Computer Graphics:Principles and Practices. Addison-Wesley.

[10] Ioannis Pantazopoulos and Spyros Tzafestas. (2002). Occlusion Culling Algorithms: A Comprehensive Survey. Journal of Intelligent and Robotic Systems Volume 35, Issue 2, pp 123–156.

[11] Joselli, M. P. (2012). A flocking boids simulation and optimization structure for mobile multicore architectures. SBGames 2012.

[12] Jur van den Berg, S. P. (n.d.). Interactive Navigation of Individual Agents in Crowded Environments. Retrieved October 6, 2016, from http://gamma.cs.unc.edu/RVO/NAVIGATE/

[13] M.Sajwan, D. S. (2014). Flocking Behaviour Simulation : Explanation and Enhancements in Boids Algorithm. International Journal of Computer Science and Information Technologies, 1-6.

[14] Mark Joselli, E. B. (2009). A neighborhood grid data structure for massive 3d crowd simulation on gpu. Games and Digital Entertainment (SBGAMES), 2009 VIII Brazilian Symposium on, (pp. 121-131).

[15] Microsoft. (2017). QueryPerformanceCounter function. Retrieved March 3, 2017, from https://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx

[16] Rahul Narain, A. G. (2009). Aggregate dynamics for dense crowd simulation. ACM Transactions on Graphics, 28(5). Proceedings of ACM SIGGRAPH Asia 2009, SIGGRAPH Asia '09. Yokohama: SIGGRAPH Asia '09.

[17] Reynolds Craig W. (2000). Interaction with Groups of Autonomous Characters. Interaction with Groups of Autonomous Characters, in the proceedings of Game Developers Conference (pp. 449-460). San Fransisco, Carlifonia: CMP Game Media Group .

[18] Reynolds, C. W. (1987). Flocks, Herds, and Schools:A Distributed Behaviral Model. ACM SIGGRAPH '87 Conference Proceedings, Anaheim, California, July 1987 (pp. 25-34). Anaheim, California: ACM SIGGRAPH.

[19] Weiss Robin M. (2010, May). GPU-Accelerated Data Mining with Swarm Intelligence. Honor Thesis. Macalester College.

[20] YILMAZ, E. (n.d.). Massive crowd simulation with parallel processing. Retrieved 09 25, 2016, from citeseerx.ist.psu.edu: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.465.7297&rep=rep1&type=pdf

