Spatial Indexing in Agent-based Crowd Simulation

Nasri Bin Othman, Linbo Luo, Wentong Cai, Michael Lees
Parallel & Distributed Computing Centre
School of Computer Engineering
Nanyang Technological University
Singapore 639798
nasr0005@e.ntu.edu.sg, {lbluo, aswtcai, mhlees}@ntu.edu.sg

ABSTRACT

Crowd modeling and simulation has become a critical tool for understanding crowds and predicting their behaviours. This is accomplished by modelling the characteristics and behaviours of large groups of people, as well as their interactions. Agent-based crowd simulation may involve thousands of complex agents interacting in sophisticated ways, in close spatial proximity, with each other. A key challenge to the development of agent-based crowd simulation is the inherent complexity that is required to model all its necessary elements. A necessary feature of such a simulation is spatial indexing. This refers to the use of data structures to organize collections of simulation entities (e.g., agents and obstacles) in a manner which allows for efficient spatial querying. This is especially pertinent for large-scale crowd simulation with agents that sense their surrounding environment periodically, as the cost of spatial querying becomes computationally expensive if done naively. In this paper, we will describe our experience in improving the existing grid-based spatial indexing approach in our agent-based crowd simulation. Also, we will explore the integration of the adaptive spatial indices (e.g., R-tree and quad-tree) into our system. The performances of various spatial indexing techniques are examined in terms of efficiency and scalability.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design studies, Modeling techniques; I.6.3 [**Simulation and Modeling**]: Applications; I.6.6 [**Simulation and Modeling**]: Simulation Output Analysis

General Terms

Algorithms, Measurement, Performance

Keywords

Agent-based simulation, crowd simulation, spatial indexing, performance analysis

1. INTRODUCTION

Human crowds are highly complex in nature. They often involve dynamic interactions among individuals and social groups. The problem of how to predict and control crowd behaviour has attracted significant interest from researchers, government agencies, safety engineers, and military personnel. However, due to the complexity of crowd, it is difficult to study the crowd behaviors based on conventional empirical analysis. Recently, crowd modeling and simulation has become a critical tool for understanding crowds and predicting their behaviours. Compared with conventional empirical research, the computer-based simulation overcomes the physical constraint of setting up controlled experiments in crowd behavior study. A synthetic virtual environment can be created and the modeling and simulation techniques can be applied to emulate various crowd events and people involved. Therefore, crowd simulation offers great potential to be used for investigating crowd behaviours in a cost-effective manner.

The agent-based approach is currently the common modeling paradigm for crowd simulation, as it provides the capability to model complex decision making process of individuals and incorporate various behavioral details into the crowd model [14]. In general, agent-based crowd simulation treats individuals in a crowd as heterogeneous and autonomous agents whose interactions yield the collective crowd behaviour. Such kinds of simulation usually involve thousands of complex agents interacting in sophisticated ways, in close spatial proximity, with each other. A key challenge to the development of agent-based crowd simulation is the inherent complexity that is required to model all its necessary elements. While numerous models [11, 15, 17, 18, 21] have been proposed to capture various crowd behavioral factors in agent-based crowd simulation, there is little work which fully investigates how the underlying environment representation and spatial querying mechanisms could affect the performance of crowd simulation.

Agents in crowd simulation are geographically distributed in a virtual environment. They are constantly moving around the environment based on their desires, beliefs and intentions. They also need to sense the nearby agents and objects in order to make decisions and interact with each other. A necessary feature of agent-based crowd simulation is spatial indexing. This refers to the use of data structures to organize collections of simulation entities (e.g., agents and obstacles) in a manner which allows for efficient spatial querying. This is especially pertinent for large-scale crowd simulation with agents that sense their surrounding environment periodically, as the cost of spatial querying becomes computationally expensive if done naively (e.g., by checking against every other entity in the whole simulation). Clearly, efficiency of spatial indexing can greatly affect the execution speed and scalability of simulation. Thus, it is one of the fundamental issues that need to be addressed for agent-based crowd simulation.

In this paper, we aim to thoroughly examine how different spatial indexing techniques and related environment representations could impact the performance of agent-based crowd simulation. To this end, we first review an existing agent-based crowd simulation system, developed in our previous work [6]. Specifically, we describe the virtual environment representation, as well as agent and object modeling in our existing system. Then, we explore how improvement on spatial indexing could be achieved in an agent-based crowd simulation system, such as in our existing system. We propose a crystallization algorithm for obstacle optimization. We also compare and implement various data structures (i.e., gird, R-tree, and quad-tree) for spatial indices. Lastly, we examine the performance of the implemented spatial indexing structures in terms of efficiency and scalability.

The remainder of this paper is organized as follows. In section 2, we review the design of our existing agent-based crowd simulation system. Section 3 provides a detailed description of obstacle optimization algorithm and how various data structures could be used for optimizing spatial indices. In section 4, we present our performance analysis, which include the agent and obstacle index performance study, as well as the scalability analysis. Finally, section 5 concludes this paper.

2. ENVIRONMENT REPRESENTATION IN COSMOS

2.1 COSMOS

In our previous work [6], we have developed a generic CrOwd Simulation for Military OperationS (COSMOS) system through the integration of game AI, distributed simulation technologies, and computer graphics and animation. The COSMOS system provides a complete suite of agent-based crowd simulation technologies integrated into a single system, powered by the Repast [16] agent-based simulation engine. In addition, crowd behavior models [2, 13] were designed to model crowd behaviour and social phenomena, e.g., emotion contagion in crowds. Crowd experiments have been conducted in crowd control and humanitarian relief scenarios [13].

The entire COSMOS architecture consists of three major components: 1) environment construction, 2) simulation, and 3) 3D visualization and animation. Environment construction is concerned with the specification of the crowd simulation scenario, and includes 3D environment construction, environment denotation, and XML mission specification. The simulation is concerned with running the crowd simulation scenario, and includes aspects such as the model and scheduling engine, the agents and entities in the scenario, the virtual environment, and the GUI. The simulation is constructed with the specifications given by the environment construction. 3D visualization & animation is concerned with visualizing the simulation on a 3D environment.

In what follows, we will briefly describe the components in COSMOS which are most pertinent to this work. In particular, we will explore how the environment is modelled, and the modeling of the agents and environmental objects in COSMOS.

2.2 Virtual environment in COSMOS

The virtual environment in COSMOS crowd simulation can contain multiple planes. Each plan presents a part of the whole environment (e.g., a storey in a subway station). Each plan is composed of specialized data structures to store different information, as shown in Figure 1. Each plane consists of several maps, as described below.

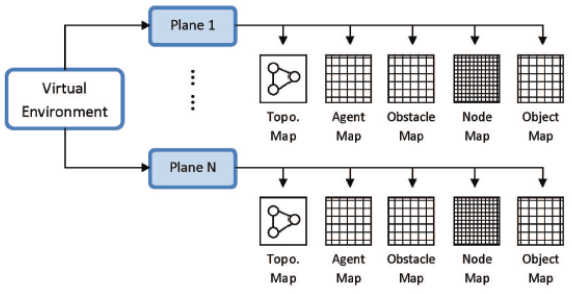

Figure 1: Architecture of virtual environment in existing COSMOS framework

Topological Map. A graph data structure to store topological areas and their relationships (i.e., connectivity). It is mainly for high-level retrieval of regional information

Agent Map. A grid map to store agents in the simulation. It consists of a 2-dimensional matrix of square grid cells, which are each identified by their grid coordinates. Each grid cell may contain multiple agents. To determine the grid cell to place an agent, its center location, which is given in real-valued 3D coordinates, must first be converted into grid coordinates.

Obstacle Map. A grid map to store obstacles in the simulation. It consists of a 2-dimensional matrix of square grid cells, which are each identified by their grid coordinates. Each grid cell may contain multiple obstacles. Obstacles are primarily used for collision-avoidance by the agents.

Object Map. A grid map to store environment objects such as Gates, Exits and Signboards. It consists of a 2-dimensional matrix of square grid cells, which are each identified by their grid coordinates. Each grid cell may contain multiple environment objects.

Node Map. A framed quadtree data structure to store accessible regions in the simulation, to be used for path planning.

By utilizing differentiated structures to store the information in the virtual environment, better efficiency is achieved. For example, to perform a spatial query on nearby environment objects, it is sufficient to consult just the Object Map for this information. This can be contrasted with a general-purpose database which contains all types of entities, whereby the spatial query has to perform post-filtering to only select environment objects.

2.3 Agents and environment objects in COSMOS

Entities in the simulation can be classified into two broad categories: human agents and environmental objects. Human agents are entities which simulate the humans in the scenario. They can observe the surrounding environment, make decisions, and act autonomously. They can also interact with other agents and objects, and exert influence on these objects (i.e., change the state of others). Every human agent in COSMOS is composed of the four basic cognitive components: perception, decision, action and working memory. The perception system (PS) retrieves information from the virtual environment, and filters interested information. The decision system (DS) makes decisions for the agent based on the information from the PS and the current working memory. The action system (AS) performs actions (e.g., goal seeking, searching) based on the decisions made in the DS and the percepts from the PS. The working memory (WM) stores predefined/persistent agent states (such as beliefs). Together, these four cognitive components comprise the "nervous system" of the agent.

Environmental objects are static entities that are parts of the environment. Environment objects can be classified according to two properties: intelligence and initiative [12]. Environmental objects which possess some degree of "intelligence" are considered smart objects, as opposed to normal objects. Smart objects are able to autonomously observe the surrounding environment and react to environmental objects. On the other hand, normal objects are unable to change their internal states autonomously. Environmental objects which may initiate interaction with other entities are considered active objects, as opposed to passive objects. Active objects may pro-actively initiate interaction with humans agents, and guide or decide actions to be performed by the human agents. On the other hand, passive objects do not provide information to human agents unless explicitly requested.

3. IMPROVEMENT ON SPATIAL INDEXING

3.1 Optimizing obstacle representation

Obstacles are entities in the virtual environment with a polygonal boundary which serve not only to restrict agent movement, but also to block their line of sight. For agents to react to obstacles, they must be able to sense and perceive the obstacles that are nearby. This is done by querying the spatial index for obstacles in the region surrounding the agent. In the existing COSMOS framework, obstacles are generated from a raster bitmap used to generate the plane. The bitmap is divided into fixed square blocks called nodes, which serve as the atomic cell unit of the grid representation of the plane. Obstacles are the nodes which have been defined as *inaccessible* to agents. The consequences of this representation are that all obstacles in the virtual world are square and identical in size, and that the size of an obstacle is the node size of the plane. One issue with this representation is that when the node size is small (to allow for fine spatial detail), the number of obstacles is massive.

Figure 2 demonstrates a simple map which requires over 4500 obstacles to represent, resulting in hundreds of obstacles being sensed by each agent in every perception cycle. The large number of obstacles causes the collision avoidance algorithm for the agent to slow down, as it has to check for collision against all the small obstacles surrounding the agent. This has to be repeated for each agent and for every simulation time step. This results in a simulation which takes an exorbitant amount of time to execute, despite the actual simplicity of the underlying geometry.

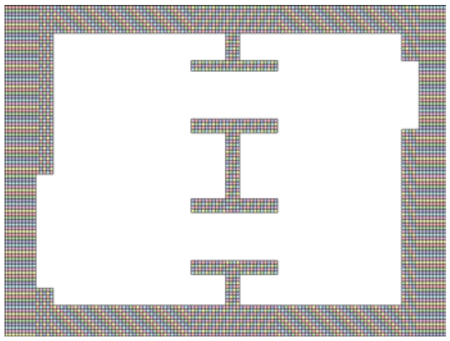

Figure 2: Example map with square obstacle representation. The obstacles are the coloured squares. The size of the map is 128 by 96 square nodes, and the true size of the map is 40 by 30 square metres. Despite its geometrical simplicity, over 4500 obstacles are created for the map. A typical agent senses several hundred obstacles in its neighbourhood.

To remedy this situation, one approach is to coalesce contiguous grid obstacles into larger obstacles, hence significantly reducing the number of obstacles in the map. The coalescing process should not only minimize the number of obstacles, but also should simultaneously minimize the geometrical complexity (i.e., minimize number of vertices) so that each obstacle can be easily processed. It would be meaningless to combine obstacles into one larger obstacle with hundreds of vertices. In this work, we propose a coalescing approach that combines adjacent obstacles into progressively larger rectangles. Rectangles are chosen as they are the simplest shape that can be formed from the union of squares. Furthermore, geometrical relational operators (e.g., *intersect*, *within*) are trivial to compute with rectangles. Thus, the problem has been reduced to generating a minimal set of non-overlapping rectangles from a 2-dimensional binary bitmap (representing accessibility).

Several algorithms exist to generate a set of non-overlapping rectangles from rectilinear figures [7, 20, 22]. However, these algorithms are not necessarily suitable in our context. This is because not only do we want to minimize the number of vertices, but we also want to minimize the aspect ratio of the rectangles detected. Hence, an easy and yet robust *randomized* algorithm is proposed: the crystallization algorithm as shown in Algorithm 1. The crystallization algorithm works similarly to crystal nucleation and growth. A random seed obstacle is selected as a seed crystal, and grown as large as possible in a greedy manner by consuming nearby uncrystallized nodes while still preserving its rectangular shape. This process is repeated until all obstacles are "crystallized".

Algorithm 1 Crystallization Algorithm for Rectangle Detection

```
 1: procedure CrystallizationDetectRect($binaryBitmap, randomEngine$)
 2:     $S \leftarrow$ GetObstaclesIn($binaryBitmap$)        ▷ $S$ is the set of unprocessed obstacles
 3:     $R \leftarrow \emptyset$        ▷ R is the set of detected rectangles
 4:     while $S$ is not empty do
 5:         $p \leftarrow$ PopRandom($S, randomEngine$)
 6:         $x \leftarrow$ NewRectangle($p.x, p.y$)        ▷ Nucleate at $p$
 7:         $D \leftarrow$ {UP,DOWN,LEFT,RIGHT}        ▷ Allowed growth directions
 8:         while $D$ is not empty do
 9:             $d \leftarrow$ PopRandom($D, randomEngine$)        ▷ Grow in random direction
10:             $x \leftarrow$ Expand($x, d, S$)        ▷ Expand rectangle as much as possible
11:         end while
12:         Insert(R,x)        ▷ Add the finished rectangle
13:     end while
14:     return $R$
15: end procedure
```

The algorithm's randomized nature reduces its chances of accidentally over-expanding greedily (i.e., local optima), but it does not guarantee a global optimal solution. Hence, the crystallization algorithm is run multiple times with different random seeds, and the best results are selected. The best result is determined by the solution which minimizes the number of rectangles generated, or by the solution which minimizes the average aspect ratio of the rectangles. An example of the result of the crystallization algorithm is shown in Figure 3. Compared to the previous map (see Figure 2), there are 300 times less obstacles. Each agent thus senses less than 10 obstacles in every perception cycle. This results in a performance boost in the simulation by at least two orders of magnitude.

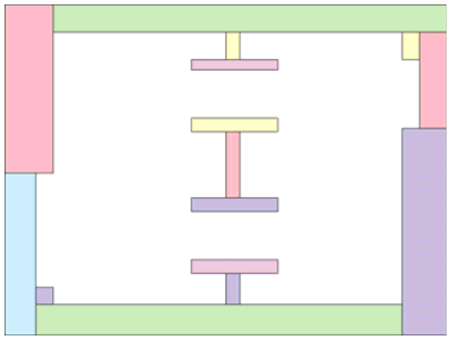

Figure 3: Example map with rectangular obstacle representation. There are only 15 obstacles, as compared to the 4500+ obstacles in Figure 2.

3.2 Optimizing spatial indices

When simulations scale to thousands of agents, the efficiency of the spatial indexes becomes a crucial factor to execution speed. In our context, spatial indices do not only perform spatial querying, but also act as containers for the entities (i.e., a spatial index is a set or multiset). Thus, not only should spatial indices be efficient at spatial querying, but they should also be efficient at set (or multiset) operations, including addition, removal, update, membership, count, traversal and query. The reason that a spatial index is adopted over a typical storage container, is that the spatial index may take advantage of the spatial characteristics of the entities to allow for efficient spatial querying. This is especially pronounced when the size of a region being queried is small compared to the extents of the whole space. By dividing the whole space into smaller and more manageable indexable regions, large performance benefits can be obtained in spatial querying by simply checking the smaller sub-regions instead of the entire space.

A spatial index can be considered to be composed of subcontainers for each indexable region of space. These subcontainers may be non-overlapping (e.g., grid [10], quadtree [8], kd-tree [5]) or overlapping (e.g., R-tree [9]); uniformly sized (e.g., grid) or non-uniformly sized (e.g., quadtree, kd-tree, R-tree); and single-level (e.g., grid), fixed multilevel (e.g., multi-level grid [4]) or hierarchical (e.g., quadtree, kd-tree or R-tree). In addition, each spatial index lends itself to specific spatial representation: either as points, axis-aligned rectangles or arbitrary shapes. However, these spatial representations can be converted easily from one to another (e.g., a point is a zero-area rectangle, a rectangle is a collection of four points). The characteristics of the subcontainers of a spatial index largely determine its suitability, robustness and performance for a given environment.

3.2.1 Grid-based index

The spatial indexing in the existing COSMOS utilizes a single-level uniform sized grid index with a fixed cell width. As described in the last section, there are three grid classes for the three types of entities in the simulation: *AgentMap* for agents, *ObjectMap* for environment objects and *ObstacleMap* for obstacles. Their underlying implementations are identical. This grid index must be instantiated with the extents of the space. The grid index uses a fixed two-dimensional map to index the grid cells, and each grid cell is an array list which stores the entities at that location. No other data structure is used for encoding the entities apart from the map of grid cells. The original grid index design is shown in Figure 4.

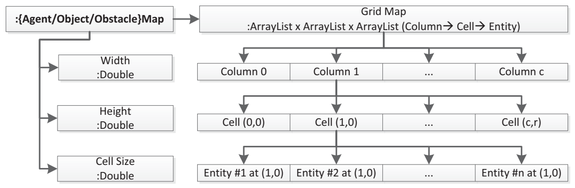

Figure 4: Original grid index design. An arrow denotes that access from the source element to the destination element is a constant-time operation.

As with typical grid indices, spatial querying on a region is reduced to computing the cells which intersect with the region and returning all the entities within. As such, spatial querying can be highly inefficient if the grid cell size is not chosen properly. If the grid cell size is too small relative to the query range, too many cells will be searched; while if the grid cell size is too large, then too many extra entities may be returned which will have to be filtered out using a *secondary filter*. The optimal grid cell size to use for a particular model depends on the size and density of the agents (taking into account heterogeneity in spatial distributions), as well as the sensor range of the agents (taking into account different agents with different sensor ranges). However, the grid cell size used in the existing implementation is fixed. Therefore, it may not be adequate in crowd simulation models.

Spatial querying using a grid index is depicted in Figure 5. A spatial query is performed by returning the contents of all cells which intersect with the query region (shaded area). The query circle is expanded to the rectangular query region, by locating the grid cell that the agent resides and expanding equally from this cell in four directions. The entities marked with + are returned, while those marked with − are excluded. Entities are represented as points in the spatial index (using centre position), which causes an entity (big triangle) to be excluded even though some part of it is within the query region. This is regarded as a false negative. False negatives are not acceptable, as they should be part of the results, but are not included. Entities which do not intersect the query circle, but intersect the rectangular query region are regarded as false positives. They are acceptable, as *secondary filtering* can be used to filter out them.

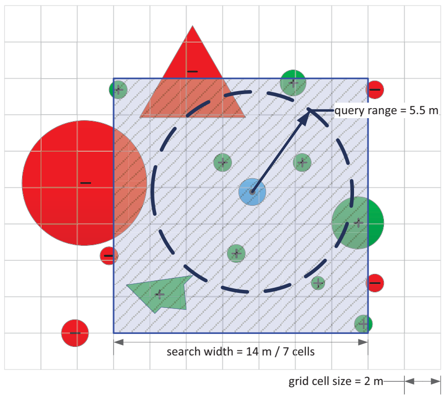

Figure 5: Spatial querying with the existing COSMOS grid index

The grid-based spatial indexing also suffers from inefficient container operations. As the only data structure used is the grid cell map, the only way to determine if an entity exists in the map is to traverse each grid cell and checking if it is contained within. Thus, while `add an entity` is a constant-time operation and `query` is potentially inefficient, other operations (such as `remove` and `count`) require the traversal of all the grid cells, and are hence highly inefficient. Thus, in models where any agent may move in a simulation time step, the entire map has to be recreated from scratch to update it. This means the entire map has to be cleared and re-populated after each simulation time step, traversing through each grid cell even if there is only one entity within it.

To improve the efficiency of the original grid index, two issues need to be addressed: 1) allowing the grid size to vary, and 2) using more efficient container operations. The first issue can be fixed easily by allowing the modeller to specify the grid size as a parameter. The second issue can be handled by creating a redundant mapping of entity-to-grid cell relations using a hash table. Thus, a constant-time lookup of the grid cell occupied by an entity is achieved. Maintenance of this hash table comes with a negligible cost as the major hash table operations are very efficient (i.e., $O(1)$ for add/remove/count/membership and $O(n)$ for traversal). In addition, the grid cells use a hash set to store the entities at that specified location, which allows for constant-time removal/membership operations as opposed to the $O(n)$ removal/membership operations of an array list. The optimized grid index design is shown in Figure 6.

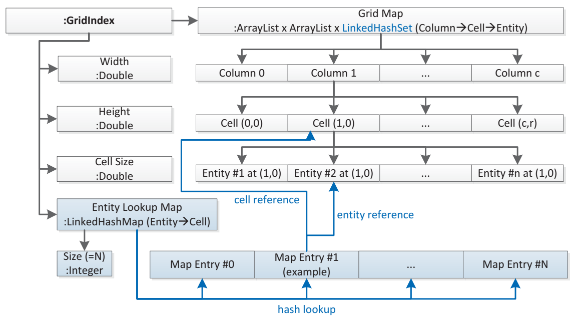

Figure 6: Optimized grid index design. The entity lookup map allows constant-time access to a stored entity and its cell, allowing for constant-time remove, update, membership, and count operations.

3.2.2 *Adaptive spatial index*

Even though the optimized grid index design can improve the performance of a simulation significantly, there is still another issue: if an entity is larger than a grid cell (as in the case of large obstacles), it may not be returned in a spatial query if its centre position is too far from the query region. Also, the optimal grid cell size is highly sensitive to the spatial distribution of the entities in the simulation (which may vary over time). To address these issues, an alternative approach is to use an *adaptive* spatial index. These are termed data-driven spatial indices [19], as their internal representations are dependent on their current contents (i.e., they are adaptive to the spatial data). They should additionally be robust enough to efficiently support most typical sequences of add/remove operations without becoming unbalanced. In this paper, we examine two such spatial indices: R-tree [9] and adaptive Quadtree [8].

An R-tree is a spatial index with the nodes representing rectangles, and the leaf nodes representing the minimum bounding rectangles of the stored entities. A parent rectangle spatially contains all the rectangles of its descendants. Sibling rectangles may overlap one another. The user may tune the maximum and minimum number of child nodes per parent node, although typical values would be robust enough for most applications. Several algorithms are proposed to improve the quality of the intermediate rectangles, including, Guttman quadratic split [9], linear splits [1] and R* [3], with the primary considerations being to reduce overlap between siblings and to minimize the aspect ratio of the rectangles. Spatial query is performed by recursively visiting the rectangles which intersect with the query envelope, starting from the root rectangle, and returning the intersecting leaf rectangles as the result.

Figure 7 shows a spatial query performed using an R-tree. A spatial query is performed by returning only the entities whose rectangles intersect with the bounding box of query circle (shaded area). The entities marked with + are returned, while those marked with − are excluded. The dashed rectangles represent the intermediary rectangles stored in the R-tree. As the query takes into consideration the size of the entities, the large triangle that was previous excluded in Figure 5 is now included. The two bottommost entities marked with + are false positives as they are outside the query circle. *Secondary filtering* is still required to filter out the false positives, however the number of false positives will always be less than or equal to the number of false positives returned by a grid spatial index.

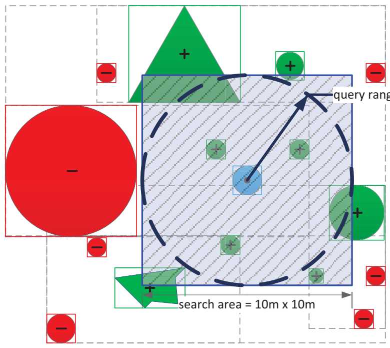

Figure 7: Spatial querying with an R-tree index

The implementation of R-trees in this work is achieved by the integration of an existing R-tree implementation, the Java Spatial Index (JSI) R-tree[1]. The JSI R-tree is a deliberately lightweight R-tree implementation using the classic Guttman quadratic split, and is tuned for very high performance. Our `JsiRtree` class uses the JSI R-tree for spatial querying, and uses a hash map (entity to spatial info mapping) to perform the container operations. Our additional hash map structure imposes a negligible cost to the index. From our experiments, without any tuning, the JSI R-tree implementation far outperforms most variants of the grid indices, except for those which are particularly well-tuned and optimized for a specific situation.

Another alternative to grid indices is also provided by the JTS Quadtree spatial index, provided by the JTS Topology Suite. A Quadtree [8] is a data structure where the nodes correspond to regions which either have 0 children (leaf nodes) or exactly 4 children nodes, with sibling nodes not overlapping each other. Each leaf node is a bucket which can consist of up to a fixed number of entities. When using a quadtree as a spatial index, dense regions are typically represented by a high amount of smaller buckets, whereas sparse regions are typically represented by a low amount of larger buckets. As such, like the R-tree, it is also adaptive to the spatial distribution of the entities. As the JTS computational geometry suite comes with a quadtree implementation, we have also adapted it into a `JtsQuadtree` class.

[1]JSI library: `http://jsi.sourceforge.net/`

4. PERFORMANCE ANALYSIS

In simulations with large numbers of agents, performance is most affected by the time taken to step through all the agents at each time step. In previous sections, we have attempted to speed up this process by examining different spatial indices and rectangular obstacle optimization. In this section, we perform several performance tests to quantify the speed improvements of our optimizations. The performance tests are executed on a dedicated HPZ400 Workstation, equipped with 3.33 GHz Intel(R) Xeon(R) W3680 processor and 12GB of RAM.

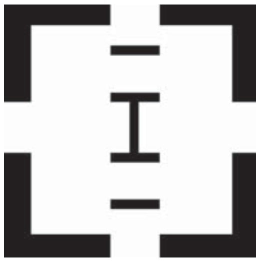

Figure 8: Tessellation tile for performance tests. Inaccessible regions are coloured black.

The test scenario consists of a virtual world with a single plane which contains a constant number of agents (predetermined for each test). These agents are light-weight simple agents whose only behaviour are to move randomly in a Gaussian random walk whilst avoiding other agents and obstacles using RVO2 [23]. Thus, the computational overhead of the agents' step function is minimized. The size of the plane is varied depending on the requirements for each test, and is composed by tessellating a 40 m x 40 m tile containing pre-defined obstacles as shown in Figure 8. The agents are placed in a uniformly random manner in the accessible regions of the plane (i.e., positions unoccupied by obstacles).

4.1 Agent index performance

The agent index test serves to compare the performance of the various spatial indices to be used for spatially indexing agents. The simulation consists of 1000 agents in a 400 m x 400 m empty plane and runs until the 100th iteration. The execution time for 10 runs are measured, and then averaged to yield the mean execution time. The test configurations include the JSI R-tree, JTS Quadtree and optimized COSMOS Grid spatial indices. Both the JSI R-tree and JTS Quadtree configurations use default parameters for construction. The COSMOS Grid spatial index is optimized to allow a variable grid cell size and has a backing map for fast container operations (see Section 3). As the grid cell size is variable, various grid cell sizes are tested.

The results of the tests are shown in Table 1. The number suffixing each Grid index entry is its grid cell size. Only the top results as well as some selected results are shown. As expected, the JSI R-tree has the best performance out of all the agent indices. Surprisingly, some of the grid indices are capable of matching its performance. However, we note that in this scenario, the spatial distribution of the agents is largely uniform, and there are only 1000 agents in the simulation. In situations where the spatial distribution is non-uniform and when there are very large numbers of agents,

Configuration	Mean execution time (s)
JSI R-tree	1.77065
Grid-120.0	1.80456
Grid-125.0	1.81590
Grid-130.0	1.86563
Grid-135.0	1.97169
Grid-140.0	1.98067
Grid-145.0	1.99724
Grid-40.0	2.01829
Grid-150.0	2.02941
Grid-25.0	2.05289
...	...
JTS Quadtree	2.71328
...	...
Grid-400.0	10.68441
Grid-2.0 (original value)	19.54321
Grid-1.0	72.14597

Table 1: Agent index performance results

Configuration	Mean execution time (s)
JTS Quadtree (Rectangular)	0.00645
JSI R-tree (Rectangular)	0.00711
Grid-120.0	0.37114
JTS Quadtree (Square)	0.39395
Grid-45.0	0.40276
JSI R-tree (Square)	0.41771
Grid-50.0	0.41892
Grid-125.0	0.43475
Grid-130.0	0.46321
Grid-55.0	0.46550
...	...
Grid-2.0 (original value)	0.49082
...	...
Grid-1.0	0.51954
...	...
Grid-400.0	1.79299

Table 2: Obstacle index performance results

we believe the grid indices will have difficulty matching the performance of the JSI R-tree.

4.2 Obstacle index performance

The obstacle index test serves to compare the performance of the various spatial indices to be used for spatially indexing obstacles. The simulation consists of **1 agent** in a 400 m x 400 m plane composed of 100 tessellated tiles in total, and runs until the 100th iteration. The execution time for 10 runs are measured, and then averaged to yield the mean execution time. Similar to agent index test, the test configurations include the JSI R-tree, JTS Quadtree and optimized COSMOS Grid spatial indices. As the JSI R-tree and JTS Quadtree spatial indices support arbitrarily large entities, they are tested with both original square obstacles and rectangular coalesced obstacles (see Section 3). As the Grid spatial index can only properly support entities smaller than the grid cell size (as discussed in Section 3.2.1), the Grid configurations are tested with only square obstacles, and not with rectangular coalesced obstacles (which can be much larger than the grid cell size).

The results of the tests are shown in Table 2. The number suffixing each Grid index entry is its grid cell size. Only the top results as well as some selected results are shown. From these tests, we can see that using rectangular obstacles yields extremely superior performance to square obstacles. Thus, the Grid index is conclusively inferior as it does not support arbitrarily large entities. The JTS Quadtree with rectangular obstacles configuration is the fastest spatial index for storing obstacles, and is slightly faster than the JSI R-tree. However, our agent index performance shows that the JSI R-tree is generally faster than the JTS Quadtree. We believe that this may be caused by a faster query performance on the JTS Quadtree and a faster insert/update/remove performance on the JSI R-tree. Since our obstacles are static, the JTS Quadtree comes out ahead in this test. Further investigation may be required to determine if the JTS Quadtree can consistently outperform JSI R-tree for storing static geometry.

4.3 Scalability analysis

The scalability test serves to compare the scalability of the various spatial indices used for spatially indexing agents. Scalability is measured by testing the spatial indexes in simulations with increasing number of agents (from 1000 to 20000 agents). The simulation consists of agents in a plane composed of tessellated tiles, and runs until the 10th iteration. The total iterations are set to be short in this test, mainly because that when the number of agents become very large, the execution time will grow exponentially for the grid-based implementations (see from our results shown later). The execution time for 10 runs are measured, and then averaged to yield the mean execution time.

When scaling the number of agents, we both test the case when the plane area is kept constant (const area) while the number of agents scale, and the case when the agent density is kept constant (const density) while the number of agents scale. In the first case, since the area is constant, the agent density increases with increasing number of agents, and each agent has to process more neighbours. In the second case, since the area grows to keep the agent density constant, each agent will still process approximately the same number of agents. Hence, the first case is expected to yield significantly slower execution times than the second case.

The agent spatial indices to be compared for each scenario configuration are: JSI R-tree, JTS Quadtree and COSMOS Grid. Both the JSI R-tree and JTS Quadtree configurations use default parameters for construction. For the optimized COSMOS Grid, grid cell sizes ranging from 20.0 to 40.0 (in increments of 5.0) are tested. As the tests are computationally expensive, other grid cell sizes were not tested. The JSI R-tree is used for spatially indexing obstacles. Rectangular obstacles were used for maximum performance. The results for the constant area configurations are depicted in Figure 9, and the results for the constant density configurations are depicted in Figure 10.

As expected, the performance of the spatial indices are much better in the constant density configurations than in the constant area configurations. This is because with constant area, each agent performs increasing cognitive processing due to having to sense more neighbours. In all cases, the JSI R-tree appears to be superior to the other spatial indices. While Grid spatial indices remain competitive in the constant density configuration, it would appear that they are not scalable in the constant area configuration. One

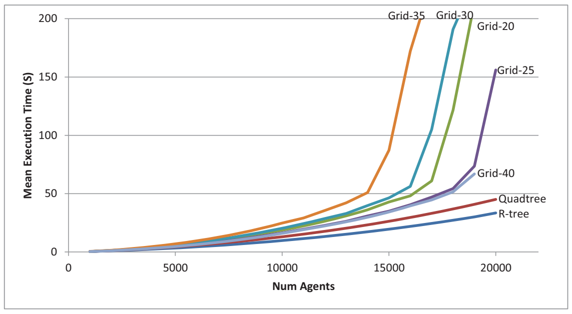

Figure 9: Scalability of agent spatial indices (constant area). Values which are too high are not shown. The best performing index is the JSI R-tree. The grid indices do not appear to be scalable with increasing agent density. At 20000 agents, Grid-35 failed with an out of memory error. As a result, we did not compute the results for Grid-40 at 20000 agents.

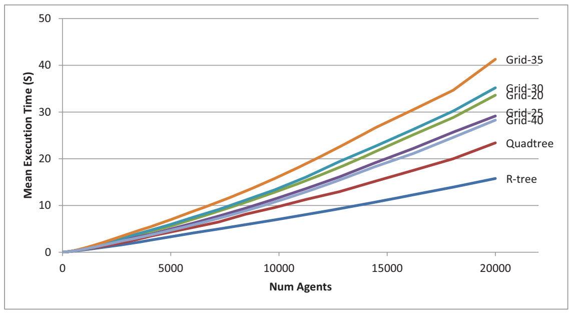

Figure 10: Scalability of agent spatial indices (constant density). The best performing index is the JSI R-tree. Furthermore, it appears to exhibit almost linear growth.

likely reason is due to the increased memory requirements of the grid spatial indices. This is demonstrated when Grid-35 failed with an out of memory error at 20000 agents.

5. CONCLUSIONS

In this paper, we have examined the spatial indexing techniques in the context of agent-based crowd simulation. Based on the existing implementation of our previous crowd simulation work, we have proposed the optimization methods for obstacle representation and grid-based index. We have also explored the integration of adaptive spatial index (i.e., R-tree and adaptive Quadtree) into the system, as compared to the grid-based index. Various performance tests have been conducted, which shows that our optimizations have not only improved the speed of simulation significantly, but also made the simulation scalable to tens of thousands of agents. From our performance analysis, it is found that the JSI R-tree implementation outperforms other implemented spatial indices in most cases. As such, R-tree is recommended for both execution performance and scalability.

It should be noted that the agents used in our performance tests are simple agents who do not perform any sophisticated reasoning. As the cognitive processing of the agents are not parallelized in our models, sophisticated cognitive reasoning is likely to add a significant performance penalty to our simulations. Therefore, further investigations are still required to investigate the performance of the agents for complex scenarios.

6. REFERENCES

[1] C. Ang and T. Tan. New linear node splitting algorithm for R-trees. In *Advances in Spatial Databases*, pages 337–349, 1997.

[2] H. Aydt, M. Lees, L. Luo, W. Cai, M. Low, and K. Sornum. A computational model of emotions for agent-based crowds in serious games. In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, pages 72–80, 2011.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *ACM SIGMOD Record*, 19(2):322–331, 1990.

[4] S. Bell, B. Diaz, F. Holroyd, and M. Jackson. Spatially referenced methods of processing raster and vector data. *Image and Vision Computing*, 1(4):211–220, 1983.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[6] W. Cai, S. Zhou, M. Low, F. Tian, H. Seah, D. Ong, V. Tay, B. Hamilton, L. Luo, D. Wang, K. Sornum, M. Lees, Z. Shen, D. Chen, X. Xiao, A. Liang, and M. Xiong. COSMOS: CrOwd Simulation for Military OperationS. Technical Report D6, School of Computer Engineering, Nanyang Technological University, Singapore, July 2010.

[7] R. Chadha and D. Allison. Partitioning rectilinear figures into rectangles. In *Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, pages 102–106, 1988.

[8] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[9] A. Guttman. R-trees: a dynamic index structure for spatial searching. *ACM SIGMOD Record*, 14(2):47–57, 1984.

[10] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1998.

[11] W. Koh and S. Zhou. Modeling and simulation of pedestrian behaviors in crowded places. *ACM Transactions on Modeling and Computer Simulation*, 21(3):20:1–20:23, 2011.

[12] A. Y. Liang, M. Low, M. Lees, W. Cai, and S. Zhou. A framework of intelligent environment with smart-active objects (IESAO) for flexible and efficient crowd simulation. In *Proceedings of the 2010 Spring Simulation Multiconference*, pages 19:1–19:9, 2010.

[13] L. Luo, S. Zhou, W. Cai, M. Lees, M. Low, and K. Sornum. HumDPM: a decision process model for modeling human-like behaviors in time-critical and uncertain situations. *Transactions on computational science XII*, pages 206–230, 2011.

[14] L. Luo, S. Zhou, W. Cai, M. Low, F. Tian, Y. Wang, X. Xiao, and D. Chen. Agent-based human behavior modeling for crowd simulation. *Computer Animation and Virtual Worlds*, 19(3-4):271–281, 2008.

[15] S. Musse and D. Thalmann. Hierarchical model for real time simulation of virtual human crowds. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):152–164, 2001.

[16] M. J. North, N. T. Collier, and J. R. Vos. Experiences creating three implementations of the Repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation*, 16(1):1–25, 2006.

[17] X. Pan, C. S. Han, K. Dauber, and K. H. Law. A multi-agent based framework for the simulation of human and social behaviors during emergency evacuations. *AI & Society*, 22:113–132, 2007.

[18] N. Pelechano, J. Allbeck, and N. Badler. Virtual crowds: Methods, simulation, and control. *Synthesis Lectures on Computer Graphics and Animation*, 3(1):1–176, 2008.

[19] P. Rigaux, M. Scholl, and A. Voisard. *Spatial databases: with application to GIS*. Morgan Kaufmann, 2002.

[20] R. Shantaram and J. Stewart. An algorithm to minimally decompose a rectilinear figure into rectangles (abstract only). In *Proceedings of the 15th annual conference on Computer Science*, page 361, 1987.

[21] W. Shao and D. Terzopoulos. Autonomous pedestrians. *Eurographics/ACM SIGGRAPH Symposium on Computer Animation (2005)*, pages 19–28, 2005.

[22] T. Suk, C. Höschl, and J. Flusser. Rectangular decomposition of binary images. In *Advanced Concepts for Intelligent Vision Systems*, pages 213–224, 2012.

[23] J. Van Den Berg, S. Guy, M. Lin, and D. Manocha. Reciprocal n-body collision avoidance. *Robotics Research*, pages 3–19, 2011.