DATABASE TECHNIQUES IN CROWD SIMULATIONS AND THE SCHEDULING PROBLEM IN SDF GRAPHS

By

Abhijeet Mohapatra

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
BACHELOR IN TECHNOLOGY
AT
INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR
WEST BENGAL, INDIA
MAY 2008

© Copyright by Abhijeet Mohapatra, 2008

INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR
DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

The undersigned hereby certify that they have read and recommend to the Faculty of Computer Science and Engineering for acceptance a thesis entitled "**Database Techniques in Crowd Simulations and the Scheduling Problem in SDF Graphs**" by **Abhijeet Mohapatra** in partial fulfillment of the requirements for the degree of **Bachelor in Technology**.

Dated: May 2008

Supervisor: _____
Dr. Partha Pratim Chakrabarti

INDIAN INSTITUTE OF TECHNOLOGY,
KHARAGPUR

Date: **May 2008**

Author: **Abhijeet Mohapatra**

Title: **Database Techniques in Crowd Simulations and the Scheduling Problem in SDF Graphs**

Department: **Computer Science and Engineering**

Degree: **B.Tech** Year: **2008**

Permission is herewith granted to Indian Institute of Technology, Kharagpur to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

THE AUTHOR RESERVES OTHER PUBLICATION RIGHTS, AND NEITHER THE THESIS NOR EXTENSIVE EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

THE AUTHOR ATTESTS THAT PERMISSION HAS BEEN OBTAINED FOR THE USE OF ANY COPYRIGHTED MATERIAL APPEARING IN THIS THESIS (OTHER THAN BRIEF EXCERPTS REQUIRING ONLY PROPER ACKNOWLEDGEMENT IN SCHOLARLY WRITING) AND THAT ALL SUCH USE IS CLEARLY ACKNOWLEDGED.

Table of Contents

Abstract

Crowd simulations are a popular and difficult research topic in the development of computer games. There has been a lot of work in the graphics community on realistically animating a large number of characters in close proximity. Unfortunately, these simulations are not particularly interactive. Crowds must be specified ahead of time, and cannot form as a result of emergent behavior; characters cannot enter or leave existing crowds.

The aim of this work is to show how to handle crowd simulations in an in-memory database. An adaptation of current motion planning algorithms is given where they can be written as database queries. Several specialized spatial indices are also introduced for use in optimizing these queries. Finally, this approach is supported with experiments that measure both the performance and the quality of the simulation.

This work also seeks to study the scheduling problem in Synchronous Data-flow Graphs (SDFs). The basic goal involves developing an online algorithm that results in a valid schedule for the SDF and to evaluate its performance on the basis of a fairness parameter. The study of convergence when a task-node changes its rate are also discussed.

Acknowledgements

I would like to thank my supervisor, Dr. Partha Pratim Chakrabarti, Dean SRIC, I.I.T Kharagpur, for his tremendous patience and constant support during this research. He has been always a great source of motivation for me, and I take this opportunity to express my gratitude for him for his able guidance.

I would also like to thank Dr. Johannes Gehrke, Associate Professor (Cornell University), Dr. Alan J. Demers, Senior Research Scientist (Cornell University) and Dr. Walker White, Research Associate(Cornell University) who had supervised my work on the Scalable Games Project during my summer internship in my junior year. They had been very kind in sharing their knowledge about the spatial indexing techniques from their SGL paper. I also owe them strongly for their constant help on Crowd Simulations in the course of my senior year.

I would also wish to extend my gratitude to Dr. Arijit Bishnu, Professor, Computer Science and Engineering Department, I.I.T Kharagpur who had helped me with pointers on Power Diagrams.

Of course, I am grateful to my parents for their patience and *love*. Without them this work would never have come into existence (literally).

Abhijeet Mohapatra

May 5, 2008.

Introduction

In recent years, computer games have become an interesting and challenging area of research. One particularly popular topic of research in games is that of crowd simulation [7, 14, 6, 1]. A crowd simulation is a case where a large number of game characters, or agents are in close proximity and need to react to each other. Crowd simulations are useful in games because they make the world seem more populous and alive. They are also important for realistic simulations of large numbers of people, such as emergency response simulations or military training games. A prototypical example of a crowd simulation is a gigantic battle like the ones animated in the Lord of the Rings movies, created by the software MASSIVE [10]. In this example there are a large number of combatants, and in the simulation each of them must search for and square off against an opponent. A more mundane simulation might involve traffic in a crowded airport. Each traveler in the simulation is assigned a gate for a connecting plane, and must navigate past all of the other travelers to reach it in time. There are many more examples of crowd simulations. At their most basic, they are concerned with the interaction of a large number of agents. Crowd simulations such as these are inherently difficult because of the n^2*problem.* Suppose we have a crowd with n agents. In order to determine its next action, each agent must query the state of the other agents. This is a O(n) computation for each agent, and thus processing all n

agents in the crowd is $O(n^2)$. Since games must process agent actions at a rate close to the graphics frame-rate, this severely restricts the number of agents that can be simulated in a single crowd. Recent work in the SIGGRAPH community on rendering crowd simulations uses techniques like statistical mechanics to produce random, but reasonable crowd-like behavior [1]. However, while this approach is acceptable for animation, the randomness element makes it unsuitable for games or other interactive simulations. A player controlling a large battle may want to assign specific opponents to individual combatants. Now the behavior of each agent is purposeful and cannot be simulated as a random process. Motion planning is a special case of perception in which the agent uses its query to navigate about obstacles in order to move towards a specific goal. White et. al. [16] introduced a scripting language called SGL to solve the n^2-problem for agent perception. SGL is built upon a simulation database; this is a real-time, in-memory database that combines the behavior of all of the game agents and processes them set-at-a-time as a single database query. Each frame is processed as a single database query to determine the agent actions, followed by a batch update performing all these actions. Using query rewrites and aggregate indexing techniques, SGL is able to reduce a large class of perception types from $O(n^2)$ to $O(n \log^d n)$. The queries supported by SGL are not enough to support true crowd simulations. In particular, they do not cover the types of queries necessary for handling motion planning. Therefore this work has the following contributions:

- **Motion planning is expressed as a query in a simulation database.** In Section 1 an overview of how motion planning works in modern games is given and we show how this can be integrated into the query plans specified in [16]. In particular we identify a new class of queries which we call collision avoidance

queries.

- **We introduce novel aggregate indexing techniques for processing collision avoidance.** In order to reduce the computation from $O(n^2)$ we have to keep from materializing the joins in these query plans. In Section 2 we introduce several special aggregate indices that enable us to do that. In section 3 we discuss the back off strategy to freeze units that are in a 'possible collision domain'.

- **We evaluate these techniques with a thorough experimental evaluation.** In Section 4 we evaluate the performance of our indexing techniques.

This work is influenced by a fair number of different fields which we cover in Section 5. Finally, in Section 6 we conclude with an overview of related and future work.

Chapter 1

Crowd Simulation Queries

Crowd simulations have been studied heavily in the graphics and animation community [7, 14, 6, 1]. However, this work either suffers from n^2-behavior or it simply animates a wandering crowd, and so all of the agents move about randomly. In particular, the scalable techniques do not allow each agent to have a fixed destination which it move towards. Thus while these techniques are useful for animating non-interactive crowd scenes, they are not particularly useful for interactive simulation. As far as we are aware, our approach is the first one to solve the n^2-problem for crowd simulations in a way that supports interactivity. The types of queries that we need to process in crowd scenes are very similar to the types seen in moving object databases [2, 4, 9]. However, both our queries and our data differ from them in very important ways. Moving object databases do allow for topological queries on trajectories such as when two trajectories collide but index structures for moving object databases are optimized to handle complicated trajectories that are materialized in the database. Our data objects are much simpler as each trajectory is only a line; we wish to predict the best next trajectory, and not query historical trajectories. Furthermore, the authors are aware of no results in the literature for finding the first

collision among trajectories. Standard nearest neighbor queries are insufficient for the reasons illustrated in Figure 1.1. The closest existing to technique to finding collisions on lines is an index for returning the set of all points that lie on a line. However, this index again assumes that all points to be found are materialized; we wish to find the intersection of two lines and do not materialize the points on either line. Finally, no moving object database supports queries relating to artificial potential fields. In order to simplify our discussions, we assume throughout that we are focusing on non-kinetic simulations. In these types of simulations, we do not need to worry about minute interactions between individual agents such as realistic collisions or the exact positioning of body parts. These types of simulations allow us to ignore problems with graphics and physics, and allow us to focus entirely on processing agent queries. Non-kinetic simulations are still a large and very important class of simulations, with applications in military training [10]. The airport simulation is another example. Even a battle like the one in Lord of the Rings can be modeled as a non-kinetic simulation if we abstract out the details of how combatants fight one another. In addition, we will also restrict our discussion to two dimensions, as this is often sufficient for handling non-kinetic simulations. This decision is primarily to make our indices and algorithms easier to understand. With that said, all of our techniques generalize to higher dimensions and we point this out where appropriate. In order to understand what queries must be processed in a non-kinetic simulation, we first need to understand how motion planning works in these types of games and simulations. From this we can identify exactly where the n^2-behavior occurs and how to avoid it.

1.1 Motion Planning in Games

In modern games, there are two different types of motion planning [12, 15]. The first type is pathfinding, which uses a traditional algorithm like A* to find a path from one location in the game world to another. Typically, this path is represented by a sequence of waypoints, which are chosen so that an agent can travel in a straight line between them without running into any other obstacles in the game. Unfortunately, as the game progresses, obstacles in the game may move and thus render this path invalid. Further more, if one agent is chasing another, then the path needs to be updated dynamically as the target moves. The naive solution to both these problems is to recompute a path whenever it becomes invalid. However, pathfinding algorithms such as A* are far too computationally expensive to be processed at frame-rate [12]. To solve this issue, games introduce a second type of motion planning known as steering. In steering, each agent has a goal location, and uses control-theory type algorithms to reach this location as fast as possible without hitting any other objects. These algorithms are typically modifications of those first pioneered in the robotics literature [8, 13]. While steering algorithms can handle dynamic obstacles, they use only simple heuristics to plan their movements around them. When the obstacles force an agent to take a very circuitous route towards the goal location (such as a very long wall separating the agent from its goal), they are not as effective as pathfinding in discovering the best way to go. To overcome the failings of these two types of algorithms, games combine them [12]. When an agent wishes to move, it sends a request for a path to an asynchronous pathfinding algorithm. This algorithm returns a set of waypoints to the agent which avoid all of the static, unmoving objects in the game. Once the agent receives these waypoints, it moves to each of them in

turn using a steering algorithm. The steering algorithm is recomputed every frame to guarantee that the agent reaches the waypoint while avoiding all obstacles, static and dynamic. Because the pathfinding algorithm can be evaluated asynchronously over several frames, it is not the true performance bottleneck in game motion planning. The true bottleneck is steering, which can easily exhibits n^2-behavior. As part of its steering algorithm, each agent must consider every other agent between it and its goal. It is easy to see then that when the agents all crowd together, steering is O(n) for each agent and hence $O(n^2)$ for all agents. For example, support the travelers in our airport simulation are all bottlenecked at the food court. Then each agent must fully scan the entire crowd to look for openings in order to navigate through. To solve this n^2-problem, we take the approach that was pioneered by SGL [16]. In that model, the state of the entire world is a database. The actions of all of the characters in the game have been combined into one single database query. During each time-step or *frame* of the simulation, the simulation database does the following:

- The query of the world state produces a table of effects. Each effect is keyed to one of the elements of the world state table. Furthermore, there is at most one effect per character in the world; in those cases where characters are affected by multiple actions, the database query has used rules to produce a single combined effect.

- Post-processing updates the state of each character. The post-processing system has rules for taking the effects given for each character and applying these effects to get the new state.

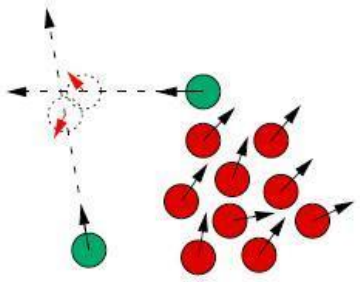

Figure 1.1:

During each frame, we loop through all the agents in the simulation For each agent, we adjust its velocity taking the following things into consideration:

- the current goal location of the object
- the location of all nearby obstacles
- if so specified, the maximum allowable acceleration (this is an optional constraint)

The purpose of the steering algorithm is to use these parameters (goal, obstacles, maximum acceleration) to choose a velocity for that agent which makes progress toward the goal location while avoiding obstacles. Once a velocity has been chosen, we move the agent forward by that amount, and back-out any collisions that might occur; steering algorithms, while useful, are not perfect. Once the agent has been moved, we proceed to the next agent and repeat the process. While collision detection and resolution is important, it has been very well studied; it is not the bottleneck in

this algorithm. Therefore, our primary focus is on the queries that produce the new velocity, which we call collision avoidance queries.

1.2 Collision Avoidance Queries

Collision avoidance queries fall roughly into two categories: outer collision avoidance and inner collision avoidance [12, 15]. Outer collision avoidance focuses on gentle correction in the face of few obstacles, while inner collision avoidance is necessary when agents and obstacles are tightly packed together. In outer collision avoidance, the agent looks ahead into the future for any possible collisions and adjusts its course if one is found by slowing down, speeding up, or moving to the side. Because the agent must have time to react, this algorithm is only useful when the possibility of collision is relatively low or many frames in the future. While this may seem unlikely in a crowd simulation, it does apply to objects on the outside perimeter, as illustrated in Figure 1.1. In this illustration, we want to prevent a collision between the agents at the top and bottom left. We do this by applying a backwards force to the lower left agent, thus slowing it down, while we accelerate and detect the upper left agent. The calculation for outer collision is fairly straightforward. For each agent we select all obstacles nearby and extrapolates their trajectory from the current velocity. We search to see if any of these trajectories intersect our own; in addition to a line intersection calculation, this is a determination as to whether the two agents will be at this intersection at roughly the same time. If any are found, we chose the least such one, and then compute a force correction based upon the two trajectories. From this formulation, it is clear where the n^2-problem arises. In a crowd simulation there are far

too many agents nearby for us to enumerate them all for this calculation. We can cut down on some of the cost of this enumeration by tightly limiting the range over which we can search for collisions. However, the more we restrict our search range, the more likely we are to experience an actual collision. Instead, we would like to have a spatial index that makes finding such collisions easy. However, as Figure 1.1 illustrates, indices tailored for nearest-neighbor queries are clearly not sufficient; we need to take the velocities as well as the positions into consideration. The right hand agents in Figure 1.1 are too close together for outer collision avoidance to be very effective. Each agent has to take all of its nearby neighbors into account when adjusting its velocity. For inner collision avoidance, games use artificial potential fields [8, 13]. In this technique, we construct a potential function that represents the geometry of the space; attractive forces represent a geometric well, while obstacles force the geometry uphill. We then compute the gradient of this potential function to find the direction of steepest descent, and adjust the velocity accordingly. This geometry is a collection of charged points. As we are limiting ourselves to two dimensions, a charged point is a tuple p = ⟨p, c⟩ where p ϵ R^2 is a point and c ϵ R is a charge. Each obstacle in our space corresponds to a charged point; the point is the position of the obstacle, and the charge is the strength of its repulsive force. As agents are themselves obstacles, they also correspond to charged points as well. Finally, the goal location of an agent is also a charged point; in this case the charge is the attractive pull of the goal. For simplicity, we fix the goal location a, and we let O be the set of charged points corresponding to obstacles. Our potential function P : $R^3 \rightarrow$ R takes an agent at point p and produces

$$P(p) = A(p, q_a) + \sum_{q \epsilon O, q \neq p} F(p, q) \tag{1.2.1}$$

Here A is a function which computes the attractive force of a, while F is a function that computes the repulsive force of each obstacle. If we fix the charge for our agent, then we can consider P to be a function of x and y. In this case, the force used to adjust the velocity of our agent is ∇P(p, a). The bottleneck in computing this gradient is not the differentiation. Avoidance algorithms are all heuristics at best, and so we do not need the exact value of ∇(p). For each point p = $\langle x_p, y_p, c\rangle$ we can approximate ∇P(p) in the obvious way by calculating P($\langle x_p, y_p, c\rangle$), P($\langle x_p + \epsilon, y_p, c\rangle$), and P($\langle x_p, y_p + \epsilon, c\rangle$) for some small value of ϵ (which is determined by the minimum charge in our geometry). So our only true bottleneck is the calculation of the potential function P. This calculation is our inner collision avoidance query. In calculating P, the n^2-behavior is readily apparent. While the attractive force A is relatively cheap to compute, we must sum up F for all nearby objects. Therefore, if we are to optimize inner collision avoidance, we should in some way efficiently compute $\sum$F(p, q) without fully materializing the join, which we address in Section 2.2 As a final word, we should note that the outer and inner collision avoidance approaches are not exclusive. Each one produces a net force on the agent to steer it away from obstacles. It is possible for us to use both techniques and sum the results together. We would particularly like to do this for agents on the perimeter of a crowd such as in Figure 1.1. However, as these queries are computed independently and then summed together, the only challenge is in optimizing them each individually.

Chapter 2

Processing Collision Avoidance

At first glance, it would seem that our collision avoidance techniques are exactly the type that moving object databases are designed for [2, 4, 9]. Unfortunately, this is not the case. Trajectory databases differ from our model in two important ways. First of all, they are designed to query a historical collection of trajectories, which are explicitly stored in the database. Much of the difficulty in indexing them is that trajectories are not straight lines and can change over time. Our model is much simpler than this. Our trajectories are rays that go on forever, as we assume will keep it current velocity unless otherwise directed. Therefore, powerful indexing techniques like TB-trees or STR-trees are not optimized to handle our types of queries. Furthermore, as we authors are aware, there has been no attempt to address our specific queries in the literature for moving object databases. This is not all that surprising for artificial potential fields. However, there does not appear to be any work in the literature on, given two trajectories, to find the first place that they collide. Trajectory databases support nearest neighbor queries (which ignore the trajectory) and queries for finding all points that trajectories cross, but never the

first place that they do. In order to prevent the queries from being $O(n^2)$, we need to keep from fully materializing the join before aggregation. The best way to do this is to push the aggregate down into the join via an aggregate index. In this section we introduce two aggregate indices for handling collision avoidance queries. Our of these first indices is a minor modification of the technique used for nearest neighbor queries. While it is not an extremely complicated algorithm, it is the first of its kind to solve this specific problem and it is surprisingly effective. Our major contribution, on the other hand, is our technique for processing artificial potential fields in inner collision avoidance. These queries are a powerful way of joining control theory and database processing.

2.1 First-to-Collide

Superficially, finding the first obstacle to collide with an agent appears to be similar to a nearest-neighbor query, for which there are many solutions such as optimized k-D trees or Voronoi diagrams [11]. Figure 1.1 shows that first-to-collide is a different problem in that we need to take into account the velocities of the moving units. In this picture, the first agent to collide with the one on the lower left is one of the furthest away. With that said, our approach will be very similar to k-D tree nearest-neighbor algorithm. In order to understand how the two techniques relate, we give a brief review and overview of how the k-D algorithm works. A k-D tree is fairly simple to understand; at each node of the tree, it splits the space along one of the dimensions to place equal numbers of points in each half; how this split is chosen is unimportant for right now. While this division cuts the space in half at every step,

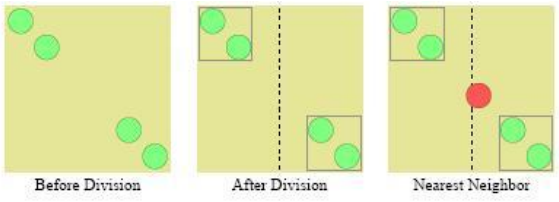

Figure 2.1: Effect of Iterations on the Priority Queue

that does not mean that the bounding boxes of the points in each half touch. The algorithm to query this structure proceeds as follows. At every stage of the query, we are are presented with two bounding boxes corresponding to the two subtrees of the current k-D node. We compute the distance d from the query point to the bounding box of each subtree. The next part of the algorithm is the use of a priority-queue. We compute d for both halves, and put them in the queue using d as the priority. Thus the front of the queue always has the closest bounding box. At each step, we pull off the closest half, and then continue the algorithm on the root node for this subtree. The search terminates when the front of the priority queue is a node with a single element. Because the priority queue can interact with the k-D tree in a arbitrary way, it is not immediately clear this is algorithm is O(n log n). In practice it is nearly so, and this is a well-studied problem related to the distribution of the points in the space[11]. We now adapt this algorithm to handle first-to-collide. For two dimensions, we are interested in the change of the bounding box over time, as shown in Figure 2.2. We approximate this with a shape that looks much like a frustrum. We attach a line in the xyt-plane to each corner of the box. The upper right corner will have a line determine by the maximum x and y components of the various velocities in the box. For example, in Figure 2.2 there is at one agent with a velocity $< 1, 0 >$ and

another with velocity $< 0, 1 >$. Therefore, we treat the upper right corner as if it were an agent moving at $< 1, 1 >$. For the lower right corner, we use the maximum x component, but minimum y, and so on for the other two corners. A big problem is that there is always a possibility of no collisions, and this possibility becomes even greater as we move up in higher dimensions. When this happens, the algorithm above will search the entire tree for each agent, resulting in an n^2 computation. The obvious solution to this problem is to enforce a time-out. If a collision is too far out in the future, there is no reason to react to it right away, as other factors may intervene to change an agent's course earlier. If the designer specifies a time after which collisions are no longer game designer can specify a time after which collisions are no longer interesting. If an agent's trajectory intersects with a frustum after this time, that subtree can be safely ignored and does not need to be added to the priority queue. We now recap our algorithm from the beginning. We start with the construction of the k-D tree. As velocity is often independent of position, we do not use any new techniques for constructing the tree; we partition the space exactly as we do in nearest neighbor. For each node in the k-D tree we construct four vectors corner_i $0 \leq i \leq 3$. We let maxX and minX be the maximum and minimum x-components of the velocity of the agents in the node, respectively. Similar we define maxY and minY. Then

$$corner_i = \begin{cases} < maxX, maxY, 1 >, & \text{i} = 0; \\ < maxX, minY, 1 >, & \text{i} = 1; \\ < minX, minY, 1 >, & \text{i} = 2; \\ < minX, maxY, 1 >, & \text{i} = 3. \end{cases} \tag{2.1.1}$$

The remainder of the algorithm is given below:

```
|Algorithm 1 Require: a an agent, T a modified k-D tree. |
1: priorityQ = {(inf,T)}
```

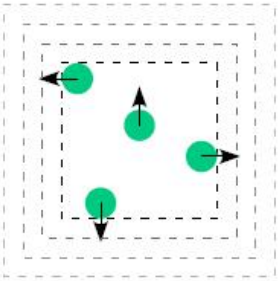

Figure 5: 2D Generalization of the k-D tree

Figure 2.2: 2D Generalization of the k-D tree

```
2: p = <a.x, a.y, 0>
3: v = <a.velx, a.vely, 1>;
4: Define the line l from p and v
5: While priorityQ not empty do
6:  remove front of priorityQ and call it front
7:  if front contains a single agent b then
8:      Compute the time t of collision of a and b if it exists
9:      if t exists and is less than CUTOFF then 10:
10:           return b
11:     end if
12: else
13: for each subtree T0 of front do
14:     d = 1
15:     for each i < 4 do
16:         Compute plane W of corner(i) and corner(i+1)
17:         Compute time t of intersection W between l
18:         d = min(d; t)
```

```
19:      end for
20:      if d < CUTOFF then
21:       Add (d; T0) to priorityQ
22:      end if
23: end for
24: end if
25: end while
```

Even with the time-out condition, it is still possible to get n^2 behavior in the worst case. Suppose we have a situation where all the agents are converging on to a single point and are due to all arrive simultaneously (the Big Crunch). The frustums are conservative, and so each agent's trajectory will intersect with a frustum before or at the same time as each agent in the frustum. As all agents intersect at the same time, this guarantees that the individual agents will all be pushed back to the end of the priority queue, and thus we end up searching the entire space in finding the first collision. In practice, pathologies like this appear to be rare. Furthermore, as we show in Section 4, they are not particularly stable and can be eliminated with small random perturbations. With that said, dealing with these unique issues for first-to-collide, the possible lack of collisions and large numbers of simultaneous collisions make it quite difficult to evaluate this index structure analytically. Therefore, we defer our analysis of it to Section 4.

2.2 Artificial Potential Fields

As mentioned above, the difficulty with computing artificial potential fields is computing $\sum$F(p, q). In order to improve the computation of this sum, we need to be more specific about what F is. Traditionally, games define

$$F(p,q) = \alpha \frac{c_p c_q}{d(p,q)} \tag{2.2.1}$$

where d(p, q) is the distance between the points p and q, and α is some scaling constant. The idea is that repulsion acts much like gravity or other properties that exhibit a power law. It is not immediately clear how to construct an aggregate index for (2.2.1). Our problem would be much more tractable if we knew that we could decompose F(p, q) into functions f1(p) and f2(q). In that case

$$F(p,q) = f_1(p) \sum f_2(q) \tag{2.2.2}$$

Therefore we would only need to compute $\sum$f2(q) once and each computation of $\sum$F(p, q) becomes a single multiplication. Clearly, this is not possible using the value of F in (2.2.1).Fortunately, it is not important that we compute this function exactly. Our primary goal is to ensure realistic steering behavior, and this function is only a heuristic. Thus if we could find an alternate function F that is just as good but satisfies (2.2.2), this would be acceptable. Our primary strategy is to use aggregate indices to give us piecewise approximations of (2) that satisfy (2.2.2) on each piece. Instead of (2.2.1), we use one of these alternate potential functions, namely,

$$F(p,q) = Ac_p c_q e^{-kd(p,q)} \tag{2.2.3}$$

as it is much easier to work with. Furthermore, on a compact interval (r1, r2) of modest size, we can chose A and k so that Ae^{-kx} is very close to α/x. Therefore this

is a reasonable choice of a potential function.

Even with this new potential function, we still do not have the property in (2.2.2). The easiest solution is to use the L1 distance, namely,

$$d(p,q) = |x_p - x_q| + |y_p - y_q| \tag{2.2.4}$$

In this case, we can express the sum as

$$\begin{aligned}
\frac{\sum F(p,q)}{A} &= e^{-k(x_p+y_p)} \sum_{x_q<x_p, y_q<y_p} e^{k(x_q+y_q)} \\
&+ e^{-k(x_p-y_p)} \sum_{x_p<x_q, y_q<y_p} e^{k(-x_q+y_q)} \\
&+ e^{-k(-x_p+y_p)} \sum_{x_q<x_p, y_p<y_q} e^{k(x_q-y_q)} \\
&+ e^{-k(-x_p-y_p)} \sum_{x_p<x_q, y_p<y_q} e^{k(-x_q-y_q)}
\end{aligned}$$

We now break up our orthogonal range query into the four different quadrants, and we thus have a different aggregate index for each quadrant. This aggregate computation is O(n log n) for the same reasons as before, except that we have four index probes instead of two. We could alternatively use a single index with four different values, the sum for each quadrant. This latter approach allows us to reduce our overhead a bit, as it needs only a single index probe. The work on artificial potential fields dates back to classic work done on robotics as far back as the 1980s [8, 13]. Its use in computer games is much more recent [12]. More recent work on artificial potential fields has been to apply it to non-spherical shapes, or to model them with fluid dynamics that allow the agents to navigate around obstacles in a much smoother fashion. Our techniques can actually be adapted to the latter work, as they involve derivatives of complex functions with a form very similar to it. The techniques used to handle artificial potential fields are very similar to recent work in using indices for function approximation [3]. Our particular technique is new and was designed

especially to handle artificial potential fields. However, it may be possible to extend it to other classes of functions as well.

Chapter 3

Backoff Strategy in Collision Avoidance Strategies

In this chapter we study the various methods to model backoff of units. The Collision Avoidance techniques using Artificial Potential fields or the First-to-collide approach may produce a scenario where a particular subset of the units may be actually colliding with one another. Here we show techniques to avoid such a configuration by identifying the collision set in O(n log n) time and freezing their motion for a later tick.

3.1 Introduction

We have considered the characterization of units as circles with a finite radii. Therefore each unit can be thought of as a tuple $< C_x, C_y, r >$. Therefore the problem of backoff can be restated as follows:

> Given n circles in the plane, find out the set of circles that do-not intersect any other circles, or equivalently the set of circles that do intersect some other circle.

We seek to tackle this problem by identifying the different cases that may arise: 1) The circles have same radii 2) The circles have different radii (in case we may assume that no circle is contained in another circle).

3.2 CASE 1: Same Radius

This case is the easiest to handle. We can easily get an O(n log n) algorithm using the traditional nearest neighbor queries on point sets.

Definition 3.2.1. Let P = p_i be the set of centers of the n circles $1 \leq i \leq n$ with radius R. Define Nbr(i) to be set of nearest neighbors of the center of circle i.

$\therefore$ The set of overlapping circles or the collision set is basically Coll = $\{i | p_i \epsilon P \wedge |Nbr(i) - p_i| < 2R\}$.

Proof. Correctness:

Say a circle C_i that is included in the collision set is not over lapping. Therefore $\forall$ circles C_j, $j \neq i$ |C_i - C_j| $\geq$ 2R. Therefore |C_i - Nbr(i)| $\geq$ 2R $\Rightarrow$ C_i is not a member of the collision set (A contradiction). An analogous argument applies to a circle that is not included in the collision set and overlapping. Using both these, it is a necessary and sufficient condition that a circle has to satisfy the membership condition in Coll to be overlapping with some other

□

```
Sketch of the Algorithm:
Let C = Set of circles.
 For all i do
```

```
 D <- FindNearestNeighbor(Ci).
    If distance (D, Ci) < 2 * R
     Output Ci.
end For
```

3.3 CASE 2: Different Radii

This is a much tougher case to handle. Power diagrams and Weighted Voronoi Diagrams [11] do help in determining the solution in this case, which however is $O(n^2)$ in the worst case. Although the worst case complexity seems unavoidable, we propose a heuristic to get around this problem, which work in O(n log n) in all cases, with certain exceptions. We also present a variant of the plane sweep algorithm that outputs the circle intersections in O(n log n) in worst case scenario.

Assumptions:

1. Circles have varied radii however no circle is contained in another circle.
2. No four centers of circles are co-circular.
3. No three centers are collinear.

3.3.1 Plane Sweep Variant

The intuitive idea behind this algorithm is that we can detect circle intersection by the intersection of the bounding boxes formed by tangents along the x and y axis. Of course there are simple cases when the circles donot intersect yet their bounding boxes may intersect. However, we can discard such false positives in $<$ O(n log n) time. Outline of the algorithm is as follows:

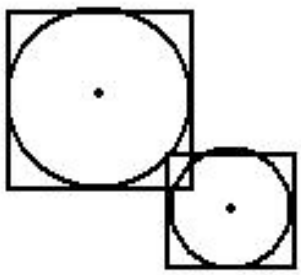

Figure 3.1: False positives in plane sweep variant

- Define the event points to be the center $< x_i, y_i >, < x_i \pm r_i, y_i >, < x_i, y_i \pm r_i >$
- Execute the traditional plane sweep algorithm with a vertical and an horizontal sweep across the event points
- Across an event point, check for ordering of the points in the event queue.
- If an intersection is detected, check for explicit intersections using the condition $|C_i - C_j| \leq R1 + R2$
- Output the circles if it passes the previous test. When the right site point of a circle is detected remove all the four points from the event queue.

Proof. If two circles intersect their bounding boxes obviously intersect, and hence this is detected when we come across some event point that determines this intersection. However, we might false positives (See figure 3.1) too in the sense that two circles may intersect What do do when we get a false positive? We just explicitly check for intersection by checking if the distance between the centers is less than sum of radii. Now, coming to the complexity, there are O(n) event points, four for every circle,

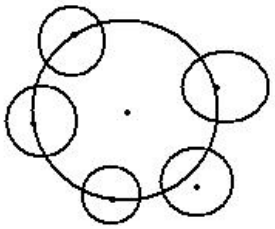

Figure 3.2: Pathological Case for the triangulation heuristic

hence the overall complexity is that of the traditional plane sweep algorithm (O (n log n)). □

3.3.2 Triangulation Heuristic

This heuristic is based on the intuition that circles are most likely to intersect with their nearest neighbors (as regard to the centers) and the assumption centers of intersecting circles are not hidden by many layers of circles in between.

Sketch of the heuristic:

- Let P = set of the n centers of the circles.
- Obtain a triangulation (Delaunay say) in O(nlogn) time
- Amongst neighboring triangles that share an edge, draw the missing diagonal to complete 4-cliques.
- Along the O(n) edges of the triangulation and the 'added edges' check for intersections.

The overall running time of this heuristic is O(n log n) and it seems very simple. This works well in most cases in experiments. However, this heuristic breaks down in the

pathological case (see figure 3.2) when we have a central circle that intersects with many other circles on its periphery each of which do not mutually intersect.

Chapter 4

Results

In this section we present experiments demonstrating the effectiveness of our indexing methods. Obviously we are interested in measuring performance, but there are other metrics as well, such as how many collisions we avoid. All of our experiments assume that the simulation database fits entirely in main memory, and that the query processor never needs to touch the disk. This is because we are comparing our indices to algorithms that are obviously $O(n^2)$. In our sample simulations, even the simplest n^2-behavior can slow a simulation of 10,000 agents down to one frame every 4 seconds. A simulation of 100,000 agents would require almost seven days to process one minute worth of movement. Therefore, the issue of how these techniques work on extremely large data sets is moot. All of these experiments were run on in MacOS X on a 2.33GHz Intel Core Duo with 3 GB of RAM. The First-to- Collide experiments were all compiled in Java; timings were done with the System.nanoTime() command. The artificial potential fields experiments, on the other hand, were compiled in gcc. These experiments were timed with the C standard library clock() command. Other, more specific, experimental parameters are explained in the appropriate sections below.

4.1 First-to-Collide

We know that there are very bad special cases that can- not be handled by our first-to-collide index. In the 'Big Crunch' example, our algorithm visits every single node in the search tree for each agent that it processes. If we use an explicit priority queue instead of other k-D tree optimizations, this means that the performance on the Big Crunch is $O(n^2 \log n)$, even worse than the naive algorithm. Our primary goal in the experiments therefore is to establish that this case is extremely rare in practice. In setting up our experiment, we started with a plane 640 X 480 in size. For each measurement, we populated the plane with agents uniformly at random. The velocity of the agents was also chosen uniformly at random, though no agent was assigned a velocity magnitude greater than 50. Finally, we chose 30 ticks as our cutoff point, so any collisions after that time was ignored. Given these parameters, we ran our first-to-collide queries on sets up to 15,000 agents. We ran our queries 200 times, and averaged the results. These results are presented in Figure 4.1. As is evident from these results, the most expensive part of our algorithm is the index construction each tick, and not the actual query cost.

One important thing to note about Figure 4.1 are the error bars. These represent twice the standard deviation computed from the runs. The standard deviations for these runs are quite large, though the worst end of each error bar well outperforms the naive algorithm. At first glance, it might seem that the performance of our index is highly variable and it is only our chosen cutoff of 30 ticks that keeps it manageable. However, we have run this experiment for other cut-offs as well, up to 200 ticks. The results that we get in each case are never statistically different from those in Figure 4.1. While the typical performance of our index seems quite good, there is

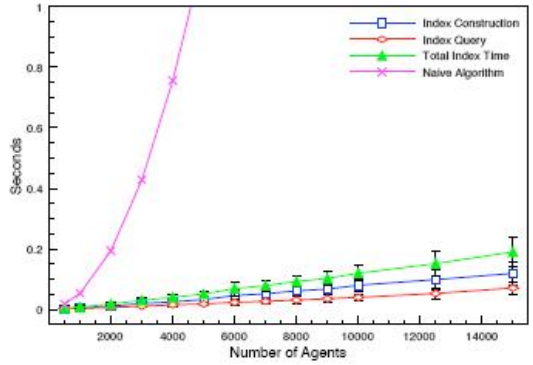

Figure 4.1: Performance of First-to-Collide Index

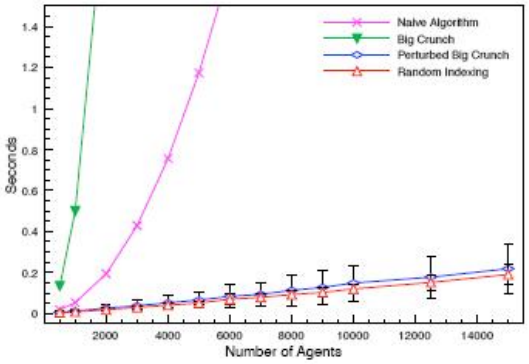

Figure 4.2: Stability of the Worst Case

still the problem of the Big Crunch. Figure 4.2 shows that the Big Crunch does indeed perform much worse than the naive algorithm. Fortunately, the Big Crunch is not particularly stable. In our second experiment, we took a Big Crunch configuration and randomly perturbed the velocities of each of the agents by a small amount (± 1 in each component); otherwise we ran our experiment exactly the same as before. As shown in Figure 4.2, this perturbation gave an enormous benefit to performance.

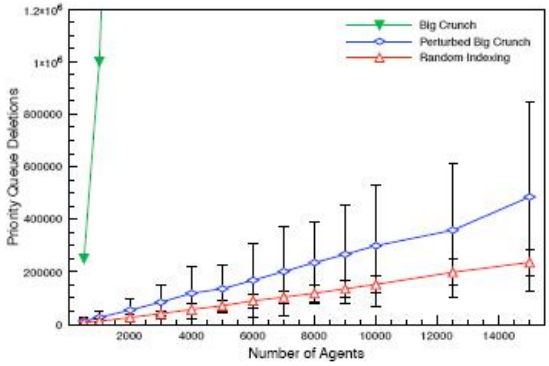

Figure 4.3: Influence of the Priority Queue

The average results and standard deviation were significantly worse than the random configurations, but still exhibited n log n-like asymptotic behavior. The extra overhead from the perturbed Big Crunch comes from the priority queue. As the Big Crunch visits every node in the index, it adds and deletes n^2 log n many elements from the priority queue over the course of the algorithm. As shown in Figure 4.3, the perturbed Big Crunch still has significantly more deletions from the priority queue than the random configuration. The wide range of variability between the various perturbed runs (which were all perturbed from exactly the same starting configuration) illustrates the difficulty in providing analytic bounds for this index structure.

4.2 Artificial Potential Fields

Unlike the First-to-Collide index, we know that our artificial potential field aggregate index is asymptotically better than the naive algorithm. There is some question about the constant overhead induced by the multiple indices and the buckets required to handle the issues with double. With that said, we know the algorithm is no worse than

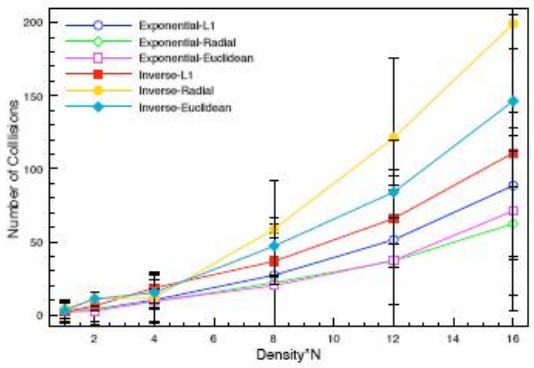

Figure 4.4: Stability of the Worst Case

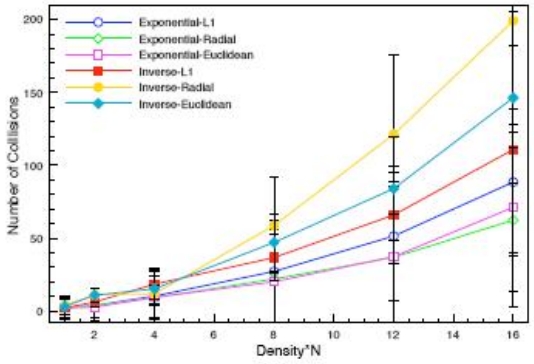

Figure 4.5: Influence of the Priority Queue

O(n log n). A more interesting question is what effect our approximation techniques have had on the artificial potential field. Our potential function is not the standard one found in games, and we are not using the traditional Euclidean distance. In order to evaluate our approximation methods, we measured the number of collisions that our approximation failed to avoid and compared this to the result for the standard potential function.

For each potential function, we discovered that the number of collisions was a

factor of the density times the number of agents. Therefore, 1000 agents uniformly distributed on a field of size 250 X 250(which has density of 0.016) suffers the same number of collisions as 20,000 agents distributed on a field of size 5000 X 5000. In this experiment, we fixed the number of agents at 1000 and varied the board size to give us different values for density X n. The agents were given a variable radius between 1 and 3, chosen at random, as well as a starting velocity between 0 and 5. In addition to the 1000 agents, we placed 24 larger, unmoving obstacles. We also generated 12 different attraction points, and assigned them to the agents like gates in an airport simulation. For each such starting configuration, we ran the simulation for 100 ticks and averaged the number of collisions. We did not resolve collisions using any post-processing; we relied on the fact that artificial potential fields are self-correcting when the area of intersection between two agents is too great. The results of this experiment are shown in Figure 4.4. Again, the error bars represent twice the standard deviation. We chose six different potential functions by varying the basic function as well as the distance metric. The inverse potential functions are those of the form in (2.2.1) while the exponential potential functions are those in (2.2.3). The L1 and Euclidean distance are self-explanatory. It is clear from Figure 4.4 that the various techniques avoid roughly same number of collisions; the standard deviations are too large to distinguish them. In our experiment, the exponential functions actually did worse at the beginning, but performed better amortized over 100 ticks as the system began to stabilize. Within each category, exponential or the inverse, there was very little difference between the L1 metric and the radial approximation. Aesthetically, there does appear to be a difference between the two when a unit is navigating around very large obstacles. However, this is very hard to

measure. As we are primarily concerned about avoiding collisions, we restricted our performance measurements to the L1 metric, which is much easier to implement. In measuring the performance, we kept the density X n constant at a value of 1. So 1000 agents were populated on a field of size 1000 X 1000 while 5000 agents filled a field of size 5000 X 5000. Unmoving obstacles were added at a ratio of 125 : 3. We used the values A = 0:929328 and k = 0:272194 in defining (2.2.3). Therefore, we use boxes of size 1000 X 1000 in order to handle the problems with double. Since we vary field size relative to the number of agents, this becomes an issue for simulations of size greater than 1000. Otherwise, the parameters were the same as our collision experiments. The results, showing the obvious success of our approach, are illustrated in Figure 4.5. These results as repeatable within one percent, and so there are no error bars. Note that even if the naive algorithm were to confine itself to buckets of size 2000 X 2000 (which is the maximum that the double will allow us to specify precisely) we still outperform it by an order of magnitude. Furthermore, the limitations of double are an artifact of current language and hardware design, and may lessen or disappear as support for large types such as long double standardizes. Therefore our results on larger sizes are still interesting.

Chapter 5

Online Scheduling of Synchronous Data Flow Graphs

5.1 Introduction

The synchronous dataflow (SDF) model [5] has been used widely as a foundation for block-diagram programming of digital signal processing (DSP) systems. In this model, as in other forms of dataflow, a program is specified by a directed graph in which the vertices, called actors, represent computations, and the edges represent FIFO queues that store data values, called tokens, as they pass between computations. The FIFO queue associated with each edge is known as a buffer. SDF imposes the restriction that the number of tokens produced and consumed by each actor is fixed and known at compile time. Each edge is annotated with the number of tokens produced (consumed) by each invocation of the source (sink) actor.

Terminology

Given an SDF edge α , we denote the source actor of α by src(α) and the sink actor α of by snk(α). We denote the number of tokens produced onto α per each invocation

of src(α) by prd(α), and similarly, we denote the number of tokens consumed from α per each invocation of snk(α) by cns(α). Each edge in a general SDF graph also has associated with it a non-negative integer delay. An actor A in a SDF G is fireable, if $\forall\alpha$ such that A = snk(α), there are sufficient number of tokens to fire A atleast once. A schedule for G is a sequence S = $f_1 f_2 f_3 \ldots$ in which each f_i is an invocation of the corresponding actor in the schedule. For each i, an invocation f_i is said to be admissible if it is fireable immediately after $f_1 \ldots f_{i-1}$ have been fired in succession. A schedule S is said to be periodic if it invokes each actor atleast once and produces no net-change in the system change. In a period, the total number of invocations of an actor A constitutes its repetition vector q_A.

5.2 Static Scheduling of SDFs

5.2.1 Computing the Repetitions Vector

The repetitions vector can be computed efficiently by applying depth-first search. An algorithm based on the one that is used in the Ptolemy system is described below:

```
procedure ComputeRepetitions(G)
    for each A in actors(G), initialize reps(A) <- 0
    select an actor A' in actors(G)
    SetReps(A', 1)
    compute x = lcm({denom(reps(A))| A in actors(G)})
    for each A in actors(G), reps(A) *= x
    for each edge e in Edges(G)
        If (reps(src(e)) * prd(e) != reps(snk(e)) * cns(e))
```

```
            error: Inconsistent Graph; exit
        EndIf


procedure SetReps(A, n)
    reps(A) = n
    for each output edge e of A
        If reps(snk(e)) = 0)
            SetReps(snk(e), ReducedFraction(n * prd(e) / cns(e)))
    for each input edge e of A
        If reps(src(e)) = 0
            SetReps(src(e), ReducedFraction(n * cns(e) / prd(e)))
```

Complexity of the above algorithm is $\Theta(|Edges(G)| + |Actors(G)|)$

5.2.2 Construction of a Static Schedule:

```
    procedure ConstructValidSchedule(G)
    1.  ComputeRepetitions(G)
    2.  for each actor in A set r(A) = min(reps(A), del(e) / cns(e))
        for-each input edge e of A.
          If (r > 0) add A to ReadyQueue.
    3.  repeat until ReadyQueue is empty
            3.1 Remove actor A from the head of ReadyQueue.
            3.2 Output (r(A) * A)
            3.3 Update the Graph and add fireable tasks to ReadyQueue
                if they are not already present.
```

5.3 Online Algorithms for SDF scheduling

5.3.1 Does Any Arbitrary Choice of a fireable actor produce a Valid Schedule?

The answer is NO. A simple counter example can demonstrate this. Consider a SDF Graph with three nodes A, B and C. The directed edges present are AB and AC with all cns() and prd() values as 1. At the beginning only A is fireable. After scheduling A, B and C become fireable. If we randomly pick a fireable node, it might be the case that we get to pick only A, B .. and ignore C which leads to an infinite buffer capacity at node C. Hence a random online algorithm with arbitrary choice of fireable actors will not produce a valid schedule.

5.3.2 An online algorithm based on number of executions of an actor

```
OnlineSchedule (G){
    Priority_Queue  P;
    // Max Priority Queue with key = max (L(i) / w(i)
    For all nodes(i) in G
        if (i is a source node) L(i) / w(i) = 1;
        else L(i) / w(i) = min j {Tokens (j -> i) / cns (j -> i)
    //Tokens(j->i) : number of tokens present in edge j->i
```

```
//cns(j->i) : number of tokens consumed from edge j->i
  on a single firing of actor i
Do{
    For all i such that (L(i) / w(i) >= 1)
       add i to P;
    node j = P.extractMax();
    Output (L(i) / w(i) * Node(j)
}While (for all i L (i) != L (i)s initial value );
```

Proof. **Case 1: Acyclic Graphs**

We prove the result by induction on number of nodes. For number of nodes = 1, the graph is trivial. For number of nodes = 2:

Consider figure 5.1(a): For source node A Li/Wi = 1. Case a = b: Obvious, the online algorithm returns a valid schedule.

Case a < b: A is scheduled first. Let k * a = b + r, such a k always exists. r < a, r < b; It is here that B is scheduled. Since its Li/Wi > 1. Then A again executes and Li of B increases. Now r accumulates in the edges as B executes. We show that finally at end of schedule this r value = 0. k' * r = a + r' for some k' where r' < a, r' < r. Now r' accumulates as B executes. Therefore, we get a converging sequence r > r' > ... which converges to 0.

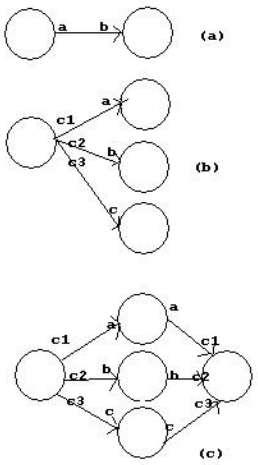

Figure 5.1: Acyclic SDFs

5.3.3 Correctness:

Case a > b: A is scheduled first. Let a = k * b + r, r < b. So 'r' remains in the Li of B when B is scheduled to execute k times. This r finally goes to 0 by a similar proof as in case a < b using b and r instead of a and r.

Now for multiple fan-out edges (figure 5.1(b)):

A is scheduled first. Either some or none (edge ratio (A/Bi) < 1) may become eligible. So each time some Bi becomes eligible it is executed and its has its own r which decreases next time it is executed till it reaches 0. Let period in which each Bi's 'r' s reached 0 be pi. So after LCM (pi) time 'r's of all Bi's reach 0. This is where schedule ends. And this is a valid schedule.

For in-degree of a node > 1: Construction required: Tie all the sources to a node (supersource). The construction ensures that A and B execute in a manner such that

finally edges AC, BC residue is 0 finally. This construction is valid for any bipartite graph which is a consistent SDF (i.e. a valid schedule exists)

Now by induction assume any general acyclic graph with n nodes is schedulable using online algorithm. Consider a graph G of size n + 1. It can be represented as a set of two subgraphs G1 and G2. Where G2 = set of sink nodes and G1 is G G2. By Induction hypothesis the online schedule of G1 is correct and the actors from which edges are directed to G2's nodes are produced proportionally. Therefore consider the graphs G1→p1, G2→p2 .. where p1, p2 ... are in G2. All these can be scheduled correctly in the online algorithm. If $|G2| = 1$ then we can adopt the proof as in case with multiple fan-ins to get the correctness proof. Therefore taken together the online algorithm produces a valid schedule.

Case 2: With Cycles Present

Refer to figure 5.2.

Case (a): fig 5.2(a). Proof is fairly straight forward. No, node in the cycle can execute indefinitely since for that to happen the nodes from which its tokens are supplied must execute at a proportional rate.

Case (b): fig 5.2(b). Node A is common to cycles C1 and C2. Again note that a cycle cannot execute indefinitely without the other executing at a proportional rate since node A is supplied by atleast 1 node from each of the cycles.

Case (c): fig 5.2(c). Edge AB is common to cycles C1 and C2. This proof is valid for any acyclic graph with one source and sink common to the cycles. Again the cycles C1 and C2 cannot individually go on executing without the other being

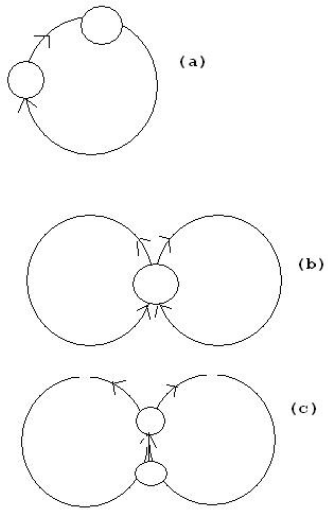

Figure 5.2: SDF with cycles present

executed since, node A which is common to both must execute and for that to happen it must receive continual supply from each its parents at the two cycles.

□

5.4 Dynamic Tasks

In this section we analyze the behavior when a given node changes its rate i.e. it becomes faster (consumes and produces at a greater rate) or slower. This involves scaling its wi's by a factor new rate / old rate. What the static scheduler does in case of such a behavior is that it recomputes the new repetitions vector for the SDF and calls ConstructValidSchedule(G) on the new Graph which is essentially the same graph with some different labels on edges. Let us now analyze the performance of the online algorithm in such a case. Consider the following situation. Given a graph G.

The online scheduling algorithm runs for some time, then a certain node-set S changes its rate. Let the new graph be G'. If this change occurs at the end of a period for G, then the online algorithm guarantees a correct schedule (by the result in previous section). So the only case remains when the change occurs at a time ¡ schedule period.

Now how does G' look like when the schedule stops in between? It is essentially G with different edge labels and different initial delays on the edges. And this delay on edge 'e' is of the form k1 * cns(e) + k2 * prd(e) i.e a linear combination of cns(e) and prd(e). So if we prove that the online algorithm gives a correct schedule for a SDF with delays then we're done. Infact this would give a stronger result regarding the convergence of the online algorithm being zero!

Proof. Consider a 2 node SDF acyclic graph. Let prd(AB) = a and cns(AB) = b. Let del(AB) = D.

L(AB) = length of queue on AB. $L(AB)_0$ = D (mod b)

$L(AB)_1$ = D + a(mod b)

$L(AB)_2$ = D + 2a(mod b)

... $L(AB)_b$ = D + ab (mod b) = D (mod b)

Now period = LCM(a,b). So after a period the Using the proof similar to acyclic SDFs we can show that the online schedule correctly schedules acyclic SDF grahps with delays.

The proof of correctness in cyclic SDF graphs also holds by similar arguments i.e after a period of schedule the Graph retains its initial delays.

□

5.5 Towards Defining a Notion of Fairness

How fair is this online algorithm as compared to others? Can we define a notion of fairness on which we can classify this algorithm? It is obvious that since the priority queue is keyed on max Li/Wi it will give more priority to tasks whose consumption rate is small.

A notion of fairness can be defined as follows: *the number scheduling decisions taken between the eligibility of the node and the actual execution of the node.* By this definition of fairness, the online algorithm has a fairness parameter of $<$ (n - 1)

Proof. Consider a node eligible for execution at time 't'. Time step here refers to a scheduling decision. Let the other nodes that are eligible be executed because of larger L_i/w_i. This cannot continue indefinitely because for that to occur, a source node (in case of acyclic graphs) or a common node (in case of overlapping cycles) must execute. L_i/w_i for a source is 1 in case of acyclic graphs so, the proof here is straight forward since the L_i/w_i of the node would be ≥ 1 after it is eligible. For the cyclic case, the case of a simple cycle is obvious, otherwise the schedule won't proceed further, which is not the case. For multiple cycles having common nodes, the common nodes will stop executing after one of their cycles has finished its quota of execution. All the other cycles must complete their share of execution before the common nodes can be eligible again and continue the schedule. Thus, the waiting time between scheduling decisions is at most n - 1. □

Chapter 6

Conclusion and Future Work

The paradigm of using database set-a-time processing for games and simulations is a very powerful one. It allows us to tackle previously intractable problems in simulations by us- ing techniques well understood by the database community. At the same time, this paradigm presents us with several unique challenges that must be solved in order to process these queries efficiently. In this work, we have introduced a special collection of queries necessary for processing motion in a simulation database. Our methods largely eliminate the n^2 bottleneck that plagues non-kinetic simulations. An interesting area of future work would be to extend these results to more complicated kinetic simulations. In these circumstances, simplifications like treating every agent as a circle or sphere are no longer valid. Solving these problems would help games and simulations provide more complex and interactive experiences over the years to come. As regards the SDF Graph schedules are concerned, the result of the convergence property of the online algorithm is very important. The static scheduler would have to recompute the repetitions vector each time to obtain the final schedule. However, the online algorithm doesn't need to recompute repetition vector. The fair scheduling

of SDFs has yet another direction of research. Note, that there exists also a not-so-obvious relation between the notion of fairness and buffer requirement of a schedule. For example if a schedule is partial to tasks whose production rate is higher then the buffer capacity increases. However a schedule that gives priority to tasks which have higher consumption rate than production rate, would tend to minimize buffer. These are however mere ideas and need to be formulated concretely. If we can establish a map between a fair-schedule with our notion of fairness and the buffer optimal schedule, then it would be possible to get bounded results on each by getting similar results on the other. That would be a big step towards getting to buffer optimal schedules which for a general SDF graph has been shown to be NP-Complete.

Bibliography

[1] S. Cooper A. Treuville and Z. Popovic, *Continuum crowds*, SIGGRAPH, 2006.

[2] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson, *Indexing moving points*, Symposium on Principles of Database Systems, 2000, pp. 175–186.

[3] S. Pope J. Gehrke B. Panda, M. Riedewald and L. Chew, *Indexing for function approximatons*, VLDB, March 2006.

[4] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis, *Nearest neighbor and reverse nearest neighbor queries for moving objects*, 2002.

[5] S. Bhattacharyya, P. Murthy, and E. Lee, *Synthesis of embedded software from synchronous dataflow specifications*, 1999.

[6] G.Still, *Crowd dynamics*, Ph.D. thesis, University of Warwick, UK, 2000.

[7] X. Tu J. Funge and D. Terzopoulos, *Cognitive modeling*, Knowledge, reasoning, and planning for intelligent characters, SIGGRAPH, 1999.

[8] P. Khosla and R. Volpe, *Superquadric artificial potentials for obstacle avoidance and approach*, IEEE Robotics and Automation (1988).

[9] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras, *On indexing mobile objects*, 1999, pp. 261–272.

[10] P. Kruszewski and M. van Lent, *Not just for combat training*, Using game technology in non-kinetic urban simulations (San Francisco, CA), Serious Game Summit, GDC, March 2007.

[11] M. Overmars M. de Berg, M. van Kreveld and O. Schwarzkopf, *Computational geometry: Algorithms and applications*, 2nd edition ed., Springer Verlag.

[12] J. O'Brien and B. Stout, *Embodied agents in dynamic worlds*, GDC, 2007.

[13] Elon Rimon and D. E. Koditschek, *Exact robot navigation using artificial potential fields*, IEEE Transactions on Robotics and Automation **8** (1992), no. 5, 501–518.

[14] Massive Software, 2007, Corporate Website.

[15] B. Stout, *Artificial potential fields for navigation and animation*, GDC, 2004.

[16] C. Koch J. Gehrke W. White, A. Demers and R. Rajagopalan, *Scaling games to epic proportions*, SIGMOD, 2007, pp. 31–42.