2012 Fourth International Conference on Computational Intelligence and Communication Networks

Multi-Agent Pathfinding System Implemented on XNA

Huang Jin
School of Software Engineering
Beijing Jiaotong University
Beijing, China
E-mail: hj34130226@126.com

Wu Wei
School of Software Engineering
Beijing Jiaotong University
Beijing, China
E-mail: wstorm910@163.com

Ling Ziyan
School of Geography
Beijing Normal University
Beijing, China
E-mail: ziyan_ling@126.com

***Abstract*—This research presents a real-time multi-agent pathfinding system for XNA game development. In our system, each agent thinks independently, and continues to search and move to reach its destination, so a group behaviors with significant individual characteristic can be better simulated; The concept of soft obstacle is introduced to implement collision avoidance between agents and crowd movement simulation; The pathfinding algorithm is applied and bridged on the 3D games, which provides efficient multi NPC pathfinding function for real-time game. The paper expounds the system from overall structure, components detail, core algorithms and practical application, and then obtains the experimental results.**

Keywords: multi-agent; pathfinding; game development; XNA

I. INTRODUCTION

Pathfinding, the problem of planning an optimal path for an agent reaching a destination without collision, is a classical problem in game development. In a static grid map, the problem can be solved by A* algorithm efficiently. However, collision avoidance between agents or agent and other dynamic obstacle must be considered, the ability of realizing the potential obstacle must be endowed to agent by improving A* algorithm. With the help of thinking independently, each agent in the multi-agent pathfinding system can accomplish the work of navigating form starting position to the goal by itself. This system also solves many problems in real-life applications, including motion planning in robotics, air traffic control, vehicle routing, disaster rescue and computer games [1].

XNA is a new game development framework based on DirectX released by Microsoft, which provides development environment for both PC and Xbox 360. This IDE (Integrated Development Environment) provides game developers with many useful tools, such as resources manager, math libraries, input, sound player, video player, internet support and so on. In addition, it offers a simple and convenient base game class that included initialization, resources import and release, update, drawing and other important method. The multi-agent pathfinding system will be applied to an existed game exploited on XNA. The discussion will focus on pathfinding algorithm and implementations of the system without involving much detail of game design.

II. RELATED WORK

A* algorithm, first proposed by Peter Hart et al. [2], is applied to find the optimal path from a starting position to a destination in static grid map. It was formed based on Dijkstra algorithm [3] that expands all nodes surrounding the starting point until it reaches a goal, however A* expands nodes in a more directed manner using a heuristic by estimating distance between the goal and the current node.

The drawback of A* algorithm, as discussed earlier, is that it can only be applied on path planning for a single target in static grid map, but not a dynamic one. To solve this problem, Silver [4] proposed an improved algorithm that widely used in video game, Local Repair A* (LRA*). In LRA*, paths are computed individually for each agent, if an agent discovers the next step will result in a conflict, it will re-plan the path to prevent the collision. However, LRA* not only often results in cyclic dependences, but also brings about the deadlock phenomenon that agents will never reach their destinations.

Jeremy et al. [5] and Jansen et al. [6] tried to estimate a direction for every grid cell that an agent may occupy, and encourage or require any agent to move in that general direction. In these algorithm, the results computed by A* or Dijkstra is translated into a direction vector for each point. These methods highly limit the paths that agents can follow, and simulate group behavior efficiently. However, this kind of algorithm cannot fulfill the requirement of game because of his weakness on expression of individual characteristic.

III. OVERALL STRUCTURE

The paper will present a multi-agent pathfinding system that can be practically employed in XNA game development by utilizing and summarizing the existing research results. The critical components of this system are given below (Figure 1).

- **Grid Map.** The grid map is used to mark static obstacles in the scene. Generally, the map is a matrix composed of 0 and 1, as 0 signs the position of barrier and 1 signs the passable area.
- **Pathfinding Unit.** The core component of the system, as it provides importance functions such as reading grid map information from text, pathfinding grid initialization, updating pathfinding gird and pathfinding method that will be invoked by any agent.

978-0-7695-4850-0/12 $26.00 © 2012 IEEE
DOI 10.1109/CICN.2012.150

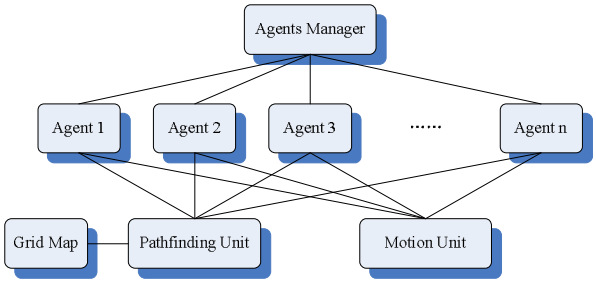

Figure 1. The Overall Structure of the system

- **Movement Unit.** The translate functions of agent in 3D world, such as moving and rotating, is implemented in this unit.
- **Agent.** The basic unit to execute pathfinding and movement behaviors in the system.
- **Agent Manager.** The tool for agent management, including registration, removal and deciding the timing and frequency that agent executes pathfinding method.

IV. COMPONENTS DETAIL

A. *Pathfinding Unit*

The pathfinding unit provides the whole system with core algorithm, which will help agents to get the optimal path. While the grid isn't static, but dynamic updates along with the changing position of obstacles or game elements, we re-initialize the pathfinding grid each time an agent invokes the pathfinding function of the unit. In A* algorithm, H represents the distant between the current node and terminal node, and G represents the cost from initial node to current node, but in our initialization of pathfinding grid, we keep H but adjust G by introducing the concept of soft obstacle, which is more correspond to the game. In addition, an eliminating list is added to the algorithm, which can mark the dynamic obstacle in pathfinding grid, so agents can't pass through them. In one word, we will search the optimal path dynamically in a grid map that composes of static obstacle, dynamic obstacle and soft obstacle.

1) *Pathfinding Grid Initialization*

Pathfinding grid is a two-dimensional array of grid cell, and important parameters in a grid cell involve F, G, H, Distance Cost and Density Cost.

Distance Cost stores the consumption of movement cost when passing the cell while Density Cost stores the consumption caused by soft obstacle. In the game, people should keep distant when the space is abundant, and avoiding a jam when the space is crowed, so the concept of soft obstacle is introduced. As is showed in Figure 2, the black box on the left is a hard obstacle that agents can't pass, and the gray circle on the right is a soft obstacle that the algorithm neither encourages nor prohibits agents pass through. Moreover, agent will receive bigger resistance as closer as it moves to the center. NPC (Non-Player-Controlled Character) and the player will be represented by the soft obstacle, and the crowd density can be controlled by modifying the radius of the soft obstacle. That is reason why we named it Density Cost.

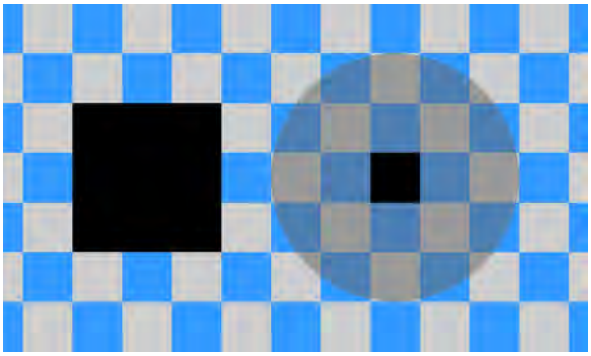

Figure 2. Hard Obstacle and Soft Obstacle

The formulate of Density Cost is

$$C_{dens} = \sum_{i=0}^{n-1} C_i \quad where \quad C_i = \begin{cases} \dfrac{r - d_i}{r} & (d_i < r) \\ 0 & (d_i >= r) \end{cases} \qquad (1)$$

C_i means the density cost from soft obstacle i, r means radius of soft obstacle, d_i means distant between the cell and center of soft obstacle i.

The formulate of G is

$$G(x) = aT(x) + bD(x) \qquad (2)$$

T means the sum of consumption from inherent distant, D means the sum of consumption from soft obstacle, a, b are weight of T and D respectively.

The pseudo-code of initialization of pathfinding grid is given below.

Create a New Empty Grid
Read Static Obstacle Information from Text

for all Grid Cell in Grid call "C" **do**
  Compare C's Coordinate with Static Obstacle Information
  if C should be Static Obstacle **then**
    Make C's Type Obstacle
  else
    Make C's Type Normal
  end if
  for all Agent in Agents of the Scene call "A" **do**
    Calculate the Density Cost by A
    Add Density Cost by A to C's Density Cost
    Calculate the Density Cost by A's Target
    Add Density Cost by A's Target to C's Density Cost
  end for
end for

for all Agent in Agents of the Scene call "A" **do**
  Make the Type of Grid Cell that in A's Position Exclusion
  Make the Type of Grid Cell that in Position of A's Target

Exclusion
end for

Firstly, an empty pathfinding grid is founded by the algorithm, and then it reads the static obstacle information from disk. Secondly, the type of grid cell is allocated according to the static obstacle information. Thirdly, the grid cells where agents stand on are all set to be soft obstacles as well as grid cells that agents will move into in the next one step. In this way, we can reduce the risk of traversing (Figure 3) and deadlock (Figure 4).

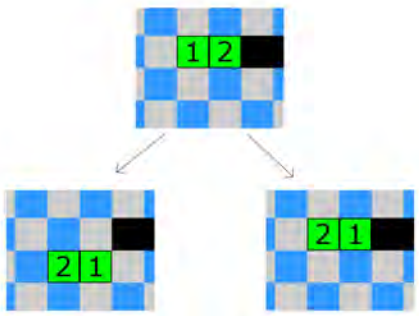

Figure 3. The illegal traversing happens in these two situations

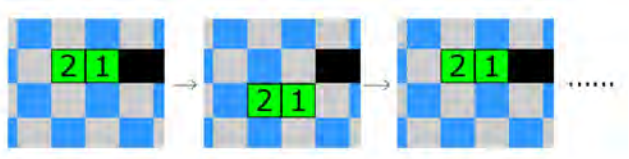

Figure 4. In some kind of situations, agent's path re-planning may results in deadlock

2) Optimal Path Searching

As soon as the initialization section is finished, we start searching the optimal path base on the latest initialized grid.

The first step of the search is to add the start node to the closed list and expand each of its neighboring cells. These cells are then added to the open list ordered by their F values. The search continues then by selecting the next node from the open list with the lowest F value. The node is removed from the open list and compared to the end node. If it is not the end node then it is added to the closed list. We then take each of its neighbours and insert them into their corresponding position in the heap. This process continues until we either reach the finishing point, or there are no nodes remaining on the open list. If we reach the finishing point then we have computed the optimal path to the end node from the start node and we can return the completed path. If our open list has become empty then there is no path from the start node to the end node.

The pseudo-code of optimal path searching is given below.

```
Create Start Node with Current Position
Add Start Node to Open List
while Open List NOT Empty do
   Update Nodes in Open List
   Sort Open List by F Value in Descending
   Get First Node from Open List call Node "N"
   Remove N from Open List
   if N is Goal then
      Found and Exit Loop
   else
      Add N to Closed List
   end if
   for all N's neighbours call "S" do
      if S is NOT in the Closed List and
      S is NOT a Static Obstacle and
      S is NOT in the Eliminating List and
      There is NO Static Barrier Corner between N and S
   then
         if S has a Copy in the Open List call "S1" then
            Renew the min G value of S1 with min {S'G, S1'G}
         else
            Add S to Open List
         end if
      end if
   end for
end while
```

Each node is updated firstly by the loop, and the F value of parent is inherited in this way. There are two additional conditions are judged as well as neglect nodes in the close list and nodes marked with obstacle, to decide whether a neighbor should be added to the open list. The first one is neglecting nodes in the eliminating list, which is a dynamic list that is setup in pathfinding grid initialization for marking all dynamic obstacles in the scene. And the second one is used to prevent the path from crossing corner of barriers. To achieve this, type of nodes signed with red dots in Figure 5 are checked and return true or false depending on situations.

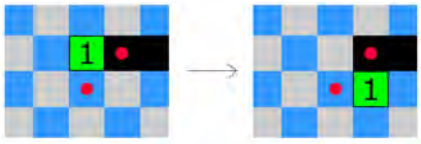

Figure 5. Agent may pass through corner of barrier without judging type of neighbours.

B. *Movement Unit*

As the application involved in this paper is a 3D game, after we finished the path searching in the 2D grid map, the agent must be ordered to excuse the movement like moving and rotating in 3D space. A movement function is provided to each agent for translation of position and direction. Moving and rotating will be performed as long as an agent calls this function at its update method.

The pseudo-code of movement function is given below.

```
Calculate the Direction from Agent to the Goal Call "Dire"
Calculate the Dot Product of Dire and Agent's Face
Direction Call "Dot"
if Dot is NOT larger than the Angle Threshold than
   if is Clockwise than
```

```
    Rotate Agent Clockwise
  else
    Rotate Agent Counterclockwise
  end if
end if

Calculate the Distance between Agent and the Goal Call "Dist"
if Dist is NOT larger than the Distance Threshold than
  Move Agent to Dire
else
  Set Path Target to Null
end if
```

Firstly, the direction of agent's destination is calculated, and then the included angle between agent's orientations and destination is got. If the included angle is larger than the angle threshold, agent twirls otherwise moves directly. We should get the right twirling direction by comparing the polar angles of the two direction vectors, and then order the agent move or not by computing the distant between agent and destination. This function will be called before the next frame is rendered.

C. *Agent and Agent Manager*

Agent is composed by two parts: pathfinding component and 3D component. Pathfinding component stores the position of agent and the latest path that successfully searches by pathfinding unit. 3D component not only stores position, direction and destination of agent, but also responsible for calling movement function at an appropriate time, making the agent move actually.

There are two maps, one is grip map for path finding, and another is real map in 3D world. The grip map is discrete while the real map is continuous, so pathfinding component and 3D component are necessary to store their information respectively. If necessary, we may need to convert them to each other.

Agent Manager is responsible for the traversal of all agents in its update function, and judging agents have finished the movement or not, if the answer is yes, then searching the optical path again, otherwise no action. Different from LPA* algorithm, we recomputed the path after the agent finished one step, instead of waits until meeting an obstacle, because it is the only way that agent can perceive the soft obstacle in the scene. If we don't execute like that, an agent will find a path in the first searching, suppose that the soft obstacles become concentrated around its path, but the path is not through the center of any soft obstacle, so even the density cost is high, the agent will still move along the first path, but this result is not what we want.

The pseudo-code of update function of agent manager is given below.

```
for all Agent in Agents of the Scene call "A" do
  if A's Path Movement Finished then
    Update Pathfinding Unit
    Find new Path for A
    Set the First Cell of the new Path A's next Path Target
  end if
end for
```

V. RESULTS

This multi-agent pathfinding system could simulate small-scale crowd (around 20 to 40 agents) finding path in a middle-size grid map (from 40*40 to 60*60). The testing routine is operated in a computer with 3.00GHz Inter Core2 Duo processor and ATI Radeon 4850 graphics card, and 45 frame rates per second is got, it will fall to 3-6 FPS when the number of agent is up to 100. Since the algorithm has not implemented in GPU, and the operating efficiency suffers obvious restrict, so in our future study, we could try to focus on transplanting the algorithm to GPU, increasing the operating efficiency that large-scale crowd could find path in a large-size grid map.

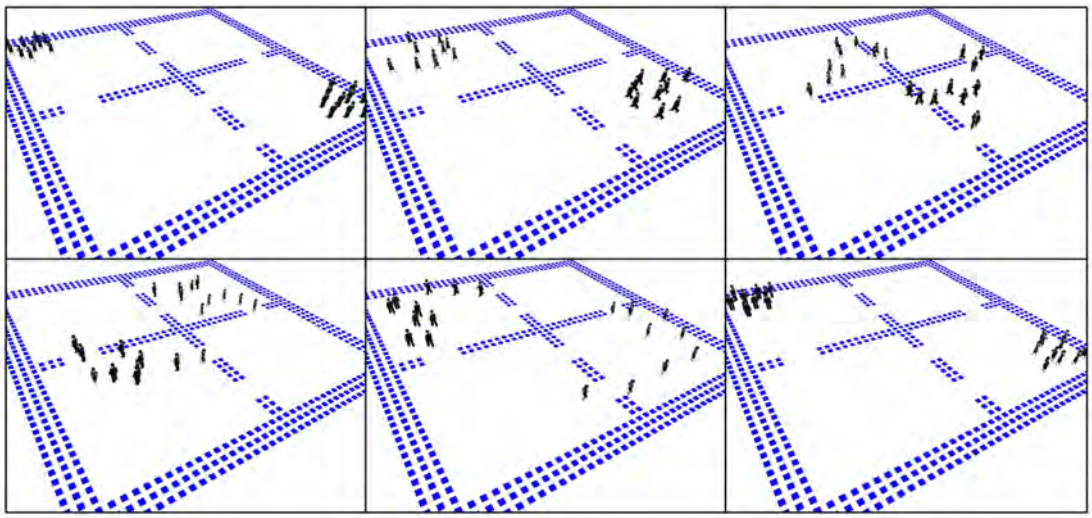

Figure 6. Pathfinding test for 20 agents

There are 20 agents finding path in Figure 6. 10 females and 10 males are separated in different corners on the map initially, and their destinations are others' initial positions. By this pathfinding system, they bypass the wall and other NPCs, and arrive at destinations successfully.

Figure 7 shows the result of this system applying in 3D game. Every NPC has their own destination in the game, they may move to any one of the rooms, talk to the player or even stop for a while, so the NPCs' behavior in the game has little group characteristic but more individual characteristic, and our pathfinding system could simulate this well.

REFERENCES

[1] Ko-Hsin Cindy Wang and Adi Botea. Fast and Memory-Efficient Multi-Agent Pathfinding. Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, 380-387, 2008.

[2] Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 4(2):100-107, 1968.

[3] Ian Millington. Articial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[4] Silver, D. Cooperative pathfinding. In Young, R. M.,and Laird, J. E., eds., AIIDE, 117–122. 2005. AAAI Press.

[5] Jeremy, Joshua, Christopher, Natalya, Shopf and Barczak et al. March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU. Advances in Real-Time Rendering in 3D Gpaphics and Games Course – SIGGRAPH, 52-101, 2008.

[6] Jansen, R. and Sturtevant, N. A new Approach to Cooperative Pathfinding. In AAMAS '08: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, 1401–1404, 2008.

Figure 7. Applying the system on a 3D game