Crowds of Moving Objects: Navigation Planning and Simulation

Julien Pettré, Helena Grillon and Daniel Thalmann

Abstract— **This paper presents a solution to interactive navigation planning and real-time simulation of a very large number of entities moving in a virtual environment. From the environment geometry analysis, we deduce a structure called *navigation graph*, which is the base to our method. After the description of this structure, we introduce a set of algorithms dedicated to answer navigation queries with a set of various solution paths and to execute the planned navigation in an efficient manner. We equally demonstrate method performance and robustness over several examples.**

I. INTRODUCTION

Robotics has made great efforts to develop motion planning methods in order to allow mechanical systems to navigate autonomously in their environment. Recently, the application field of these methods significantly expanded, such as to Biochemistry, Architecture, Ergonomics, and Computer Graphics (CG). For CG applications, motion planning increases the autonomy of digital actors, eases a user's navigation in a virtual world and solves user queries from high level directives (e.g., commanding an army in a video game) among other functionalities. A specificity of CG applications is a frequent need for interactivity: performance and robustness are main issues. We principally consider the specific case of a large number of moving entities evolving on the terrain of a given virtual environment. We have developed an architecture able to solve users' navigation queries interactively, and to update the position of each entity in real time.

A number of criteria and objectives have conditioned our technical choices. First of all, *performance*: computer power increase has doubtlessly improved computational times to those of previous approaches. However, the complexity of environments' geometric models and the number of moving entities in them has increased even more. Our method allows the abstraction of environment geometries. Then, *transferability*: our method is applicable to a large class of environments and moving entities. It is based on simple geometrical expressions and properties. It is thus easily adaptable. Finally, *scalability*: our architecture is scalable. In other words, the user can distribute the computing resources locally in space and time during simulations, and focus them where most needed.

J. Pettré performed this work at EPFL-VRlab, sponsored by Marie Curie Action, grant "RAGA" n.11166 FP6-2004-Mobility-5

J. Pettré is with IRISA-INRIA, Bunraku Team, Campus de Beaulieu, F-35042 Rennes, France julien.pettre@irisa.fr

H. Grillon and D. Thalmann are with EPFL-VRlab, CH-1015 Lausanne, Switzerland {helena.grillon, daniel.thalmann}@epfl.ch

Our method's contributions are, first, a cell-decomposition method which captures the environment's topology and geometry and is adapted to environments combining uneven terrains and multi-layered surfaces. Second, it proposes a data structure designed for both navigation planning and simulation. Third, it offers a specific navigation planning technique which searches for various solutions to a single problem. Finally, it proposes a set of algorithms to respond to user queries such as navigation, neighbor or visibility requests.

II. RELATED WORK

Navigation planning is a specific case of motion planning, [1], [2], [3], which is mainly studied by the Robotics community. However, we will once again focus on their application or development by the CG community [4]. Basically, three main classes of solutions to the motion planning problem can be distinguished: local approaches, probabilistic ones and deterministic ones.

Local approaches fit the problem of collision avoidance between two moving entities: the potential fields method [5] is very popular and has inspired many solutions [6]. However, when used for global path planning, they generate many problems: parameters tuning (naturalness vs. collision freeness), goal conflicts and local minima. ***Probabilistic approaches*** randomly explore the free configuration space [7] of a given problem, while capturing its connectivity into a roadmap. Multi-query (PRM) [8] or single query (RRT) [9] solutions exist, as well as many variants. These have been successfully applied to computer animation problems such as to the locomotion of human-like characters [10], [11], [12], [13]. Such approaches fit high-dimensional problems, with systems having numerous degrees of freedom, whilst the navigation problem is frequently lowered to a 3 or 4-dimensional problem (position and orientation of the system). Paths synthesized by PRMs require optimization which dramatically increases the solution's computational cost. Moreover, when dealing with multiple moving entities, PRM solutions tend to always produce similar paths (the ones captured by the roadmap), which increases the potential number of collisions between entities and may have an impact on path naturalness. ***Deterministic approaches*** use an exact and continuous representation of the space and of the considered system mobility. This generally results in a very complex system, making them unenforceable. Environments are therefore discretized to overpass these limitations. The use of a grid laying on the floor to capture obstacle position and navigable space is the most frequent solution. A* or Dijkstra's algorithms are used to search for solution paths [14], [15]. To optimize

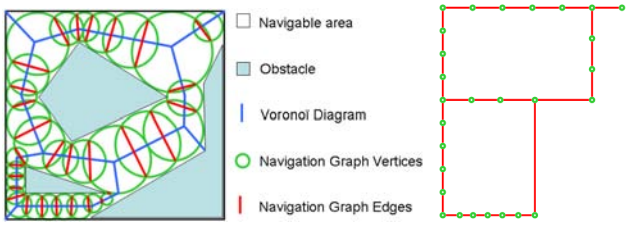

Fig. 1. *left:* Geometric representation of a $\mathcal{NG}$ in a 2D academic environment. *right:* $\mathcal{NG}$ itself for the same example.

search times, multi-resolution or hierarchical grids can be used [16], [17]. A classical drawback to these solutions is the low aspect quality of the results: jagged paths and sharp corners. A solution consists in post-processing the solution paths, which increases computation time. Our approach falls into the deterministic class by decomposing the environment into a set of 3D navigable cells. However, we compute this decomposition from a discrete space representation. Decomposition methods have already been used for crowd simulation, like in [18], and proved to be efficient. Our method improves such approaches by considering a larger class of terrains (uneven and/or multi-layered), by extending data structures to efficient navigation, neighborhood and visibility queries, and by providing scalable simulations.

III. NAVIGATION GRAPHS

A. Principle

A Navigation Graph ($\mathcal{NG}$) results from a cell decomposition of the navigable parts $\mathcal{C}_{nav}$ of a considered environment. As in classical decomposition methods, $\mathcal{NG}$ vertices are cells free of obstacle and adjacent cells are linked by an edge. The decomposition is not exact as the computation is based on a discrete representation of the obstacles (see next section). The method is dedicated to entities moving on a floor or terrain. $\mathcal{C}_{nav}$ is thus the union of the environment surfaces:

- flat enough: for a surface to be navigable, its slope must be within the bounds of a user defined limit angle,
- free of obstacle,
- with a high enough free-space above them - still according to considered entities characteristics.

Cells are cylinders laying on $\mathcal{C}_{nav}$ and are the $\mathcal{NG}$ vertices. Cells are adjacent when the corresponding cylinders overlap. Adjacency is naturally modeled by the $\mathcal{NG}$ edges. In order to capture $\mathcal{C}_{nav}$ in a compact manner, cylinders are centered on the Voronoï diagram [19] of the navigable surfaces and their center is excluded from any other cell. As for others decomposition-based techniques, a property of $\mathcal{NG}$ is that any point belonging to a given navigable cell can reach any point belonging to an adjacent cell, passing by any point of their intersection (depending on conditions). Thus, the navigation planning is reduced to a graph search problem.

Fig. 1 schematically illustrates a $\mathcal{NG}$ computed for a simple 2D environment. However, a novelty of our method is to fit 3D environments, even for those combining uneven and multi-layered surfaces.

B. $\mathcal{NG}$ computation

We previously introduced a technique to compute $\mathcal{NG}$s [20] using an intermediate grid and graphics hardware-based operators. The method consists in 5 main steps:

1) *environment geometry sampling*: we create a regular grid of points matching the environment surfaces. As multi-layered environments may be considered, a simple elevation is insufficient since several elevations may correspond to given horizontal coordinates.
2) *grid mesh*: two neighboring grid points are interconnected when the slope of the in-between space is beneath the user-defined maximum slope angle and free of obstacle. With this stage, we provide a mesh capturing $\mathcal{C}_{nav}$ in a discrete manner.
3) *clearance map*: we compute the clearance for each grid point, i.e. the distance to the nearest obstacle or high-slope. Given the the grid mesh computation method, this distance is approximated by the distance to the nearest grid point not connected to all of its direct neighbors. Indeed, The lack of connection reveals the presence of an obstacle or a high-slope.
4) $\mathcal{NG}$ *deduction*: we then use a subset of grid points to compute the $\mathcal{NG}$ vertices. A graph vertex (cylinder) is created from a grid point by using it as centre and its corresponding clearance as radius. The grid point with maximum clearance is selected to create the graph vertex and all the grid points covered by the vertex are disabled for selection. The process is then reiterated until no more grid points remain for selection. Doing so, a majority of cylinders are centered on the Voronoï diagram corresponding to the environment.
5) *visibility pre-computation*: for each $\mathcal{NG}$ vertex, we compute the visibility of all other vertices according to the four main cardinal points. We can then use this information for visibility queries between moving entities.

Some examples of environments and corresponding $\mathcal{NG}$, as well as computing times, are given in the Results section.

C. Data Structures

In order to allow the implementation of our method, and for a better understanding of the algorithms we use, we here detail the contents of our data structures. $\mathcal{NG}$ captures the following information:

1) For each vertex. *Navigable cylinder geometry*: center, radius and height. *Local elevation map*: the portion of the intermediate grid - used for $\mathcal{NG}$ computation - located under the cylinder is copied and stored. As $\mathcal{NG}$ handles multi-layered environments, this allows to solve elevation queries efficiently. *Visible vertices*: the list of all vertices that can be partially or totally seen from the current vertex. This allows to solve visibility queries efficiently. *Included moving entities*: the list of all moving entities currently navigating in the vertex. This information is managed and updated by the simulation loop. *Adjacent vertices*: the list of

vertices sharing an edge with the current vertex. This allows to solve neighbor queries efficiently.

2) For each edge. *Linked vertices*: the pair of vertices sharing this edge. *Gate geometry*: the coordinates of a line segment at the linked cylinders' intersection. *Cost*: the distance between the two linked cylinders' centers.
3) For each mobile entity. *Steering methods*: able to steer the mobile entity toward a way-point, whilst avoiding other mobile entities. Optionally, we can scale the steering method, i.e., we can change its complexity and precision with a parameter. This will be discussed in the Simulation part of the next section. *Currently crossed vertex*: in order to know in which navigable area the mobile entity is currently located. This information is managed by the simulation loop. *Currently followed path*: resulting from a navigation query. *Currently followed way-point*: which is selected along the currently followed path.

D. Discussion

As mentioned before, our method is inspired by cell-decomposition motion planning techniques. However, our method is not perfectly accurate: the decomposition does not capture the complete navigable space, and the graph computation uses a discrete representation of the environment. A high-precision grid may therefore be required in order to capture narrow passages, which results in long comuptations. Nevertheless, advantages of our method are its robustness (we have tested it on many environments whose meshes were crude from design), its ability to consider both uneven and multi-layered environments, and the lack of need for expertise to use it.

Another advantage of our method is its use of very simple geometric expressions to model the navigable space: cylinders and line segments for the graph vertices and edges respectively. This representation is not perfect in all cases: for example, long corridors require many adjacent cylinders to be captured. However, our model allows to solve basic queries efficiently. For example, a simple distance test allows to determine if a moving entity is contained in a vertex or not. It equally requires little memory for storage.

The next section illustrates the use of $\mathcal{NG}$ for crowd navigation planning.

IV. NAVIGATIO AND OTHER INTERACTIVE QUERIES

A. Navigation Planning Queries

We use 2 navigation planning algorithms, one to plan the navigation of a single entity between two locations (Alg. 1) and one to create a navigation flow between two locations, i.e., for a large number of entities all moving between identical locations (Alg. 2).

We solve Single Navigation Queries (Alg. 1) in a classical manner: given a navigation graph $\mathcal{NG}$, and two locations to join (for which the corresponding $\mathcal{NG}$ vertices v_i and v_d are found), we launch a Dijkstra's shortest path search. We deduce the resulting path from the set of edges between both

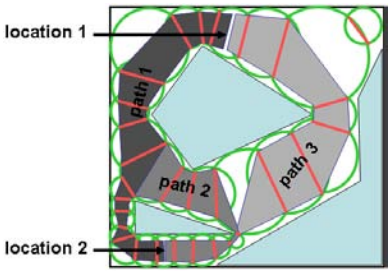

Fig. 2. Solution paths to a navigation flow query between locations 1 and 2: the proposed algorithm provides several solutions to a single query. The objective is to dispatch many entities moving between identical locations. Each entity can select its own path, and use the full width of the corridors composing the path to obtain a unique trajectory.

Algorithm 1: Single Navigation Query

Data: current location p_{init}, destination p_{dest} and $\mathcal{NG}$
Result: a solution path (set of edges)
$P_{sol} = \{e_1, ..., e_n\}$

```
begin
    v_i ← the vertex including p_init
    v_d ← the vertex including p_dest
    P_sol ← Dijkstra(v_i, v_d, NG)
end
```

locations. As edges are line segments (we also call them *gates*), we can model the path as a corridor between the two desired locations. An example of Navigation Query is illustrated in Fig. 2. This first algorithm provides *path 1* as unique solution.

Another problem is to compute a trajectory for a moving entity which is always confined to the free space. For example, if way-points are picked within each successive gate and a linear steering is used to join them, the trajectory is contained in the solution path (the solution corridor). However, in the case of mechanical constraints (e.g. non-holonomy), the user must take care of this issue.

Our objective is to address the problem of numerous entities navigating in a same environment and particularly, the case were many of them navigate between identical destinations. In this case we create a Navigation Flow between the two locations. We avoid the concentration of all entities on a same trajectory in order to limit the potential number of inter-collisions (solving interactions also has a high computational cost) and increase realism. In order to do this, we can exploit corridor width to dispatch the entities. However, this can be insufficient, especially when the gates are narrow: a congestion may appear. The second navigation planning algorithm Alg. 2 is aimed at providing a second level of variety by answering queries with a set of paths instead of a single one. As in Alg. 1, the shortest path is our first solution path. We then search for an alternative path avoiding the narrowest gate: the gate's cost is increased and Dijkstra's search is invoked again. Edge cost can only be modified once and the algorithm stops when no more edge cost can be modified. Optionally, the process stops when a

Algorithm 2: Navigation Flow Query

Data: current position p_{init}, destination p_{dest}, $\mathcal{NG}$ and optionally a max. number of paths to find N_{max}
Result: a set of solution paths $F_{sol} = \{P_{sol_1}, ..., P_{sol_n}\}$

```
1  begin
2      v_i ← the vertex including p_init
3      v_d ← the vertex including p_dest
4      E_inc ← {∅}
5      Stop ← false
6      while Stop is false and card(F_sol) < N_max do
7          Stop ← true
8          P_sol ← Dijkstra(v_i, v_d, NG)
9          if P_sol ∉ F_sol then
10             F_sol ← F_sol ⋃ {P_sol}
11         if
           ∃e \ e ← Thinnest({e | e ∈ P_sol ∧ e ∉ E_inc})
           then
12             e_cost = e_cost × 10
13             E_inc ← E_inc ⋃ {e}
14             Stop ← false
15 end
```

Algorithm 3: Simulation Loop

Data: simulation initialized
Result: updated situation

```
1  begin
2      ComputeLoS
3      for all vertex V ∈ NG do
4          if UpdateRequired then
5              for all MovingEntity M∈ V do
6                  SteeringMethod
7                  if WayPointReached then
8                      if EndOfPath then
9                          GoBackward
10                         ChooseCurrentBestPath
11                     ComputeNewWayPoint
12                     MoveToNextVertex
13     UpdatePathsCosts
14     RenderScene
15 end
```

number of solution paths is reached. Note that we developed a variant stop criterion where the relative path lengths are used: the algorithms stop when the last found path length is a times longer than the first and shortest one, $a \in [1..\infty]$.

At the end of the planning stage, the data structure is completed: the entities are placed at their initial location and each vertex's list of *included moving entities* is setup accordingly. The *currently crossed vertex, followed path and way-point* are listed for each entity as well. The real-time simulation, i.e., the iterative udpate of the situation for each entity according to the results of the planning stage then starts.

B. Simulation

The objectives of our simulation loop are specific. We want the highest performance possible in order to allow real-time updates ($25Hz$ at least) along with visualization. In our case, we look for believability. A spectator observing the scene with moving entities should be presented with a simulation of the best quality level on the foreground. For background areas, we want the entities to achieve their goal, but assumptions and simplifications are permitted.

Algorithm 3 is looped to update the simulation. Whereas classical solutions loop over each moving entity, a specificity of our simulation is to consider each area successively. These areas are delimited by the $\mathcal{NG}$ vertices. Before update (line 2), Levels of Simulation (LoS) are computed. A LoS is a score assigned to each $\mathcal{NG}$ vertex which depends on the the point of view location (view centrality and distance). Vertices having a high LoS are close to the point of view and located in the central part of the screen. Vertices having low LoS are far from the point of view, on the borders of the screen or even invisible.

When a vertex is visited, update is required or not (line 4), depending on the LoS. If the LoS is high, update is required frequently ($25Hz$). On the contrary, if the LoS is low, the udpate is done at lower frequencies (from 1 to $15Hz$).

When update is required, the position of each moving entity included in the vertex is updated. Once again, update quality depends on the LoS (line 6). For low LoS, entities are steered in a simplified manner: we use linear steering and do not consider collisions between entities since we assume they are not detectable at a far distance and even less in invisible areas. For high LoS, we steer entities using smooth trajectories and velocity accelerations. We equally take into account inter-collisions by using Reynolds' [21] steering method.

Steering methods require way-points to lead the entities along the followed corridors. For a given entity, once the tracked way-point is reached, a new one is picked within the next gate to be crossed (line 7). We use an individual parameter to compute the way-point (line 11), so that the entity crosses a gate always on the same side (the parameter continuously changes from 0 to 1 while the way-point position in the gate line segment continuously moves from left to right). The moment a gate has just been crossed also corresponds to a change of area for the entity. The vertices *included moving entities* lists are updated accordingly (line 12), as well as the other entity-related variables. Doing so, we always know where a given entity is, and which entities are in a given place. This consists in crucial information in order to solve interactive queries, presented in the next section.

If the end of the path is reached, the entity turns around and is sent back to its original location. Entities go back and forth indefinitely. However, new goals can be assigned

interactively. A different path can be assigned to the entity, within the set of solutions provided by Alg. 2. The best solution is not necessarily the shortest one. We consider the current occupation of each path to compute the best solution. We take into account both the distance to a given edge and the local population density to compute an average travel time. The lowest travel time path is selected and assigned to the entity. The paths' travel times are recomputed at the end of the update loop (line 13) if necessary: as it is not a highly dynamic variable, it can be updated at low rates. Finally, the situation is rendered to the screen, however, this is not this paper's main issue.

C. Neighbor and Visibility Queries

During the simulation we use neighbor queries in order to solve collisions between moving entities. Alg. 4 allows to compute the list of moving entities potentially in contact with a considered entity E. As each vertex stores the list of entities currently included in it, the complete list can be computed with a limited number of distance tests. Note that adjacent vertices are visited since they may contain close enough entities. Recursively, the search may be extended to other linked vertices, at a deeper level, if neighboring entities at a farther distance are to be searched for.

Algorithm 4: Neighbor Query

```
   Data: an entity E
   Result: list of entities L_neighb potentially in collision
1  begin
2      V_E ← E :: current vertex
3      for all entity e_i ∈ V_E :: included entities do
4          if DistanceTest(E, e_i) then
5              Add e_i to L_neighb
6      for all vertex v_i ∈ V_E :: linked vertices do
7          for all entity e_i ∈ v_i :: included entities do
8              if DistanceTest(E, e_i) then
9                  Add e_i to L_neighb
10 end
```

Each $\mathcal{NG}$ vertex equally refers to a list of visible areas. A list of visible entities to a given entity E can be computed with a limited number of tests (Alg. 5). Such queries may remain too complex to be solved interactively in the case of complex environments and high density areas. Indeed, visibility tests (line 5) complexity depend on the number of triangles composing the environment. Tests may be done by casting rays (using a collision checker) or by occlusion tests using OpenGL extensions.

V. Results

We have applied our technique to crowds of virtual pedestrians. Fig. 3 illustrates some of the tested environments, with snapshots of the computed $\mathcal{NG}$. Outcomes to Navigation Queries are also shown. The Stonehenge-like environment is representative: the presence of pillars in the middle of

Algorithm 5: Visibility Query

```
   Data: an entity E
   Result: the list of all visible entities L_vis
1  begin
2      V_E ← E :: current vertex
3      for all vertex v_i ∈ V_E :: visible vertices do
4          for all entity e_i ∈ v_i :: included entities do
5              if VisibilityTest(E, e_i) then
6                  Add e_i to L_vis
7  end
```

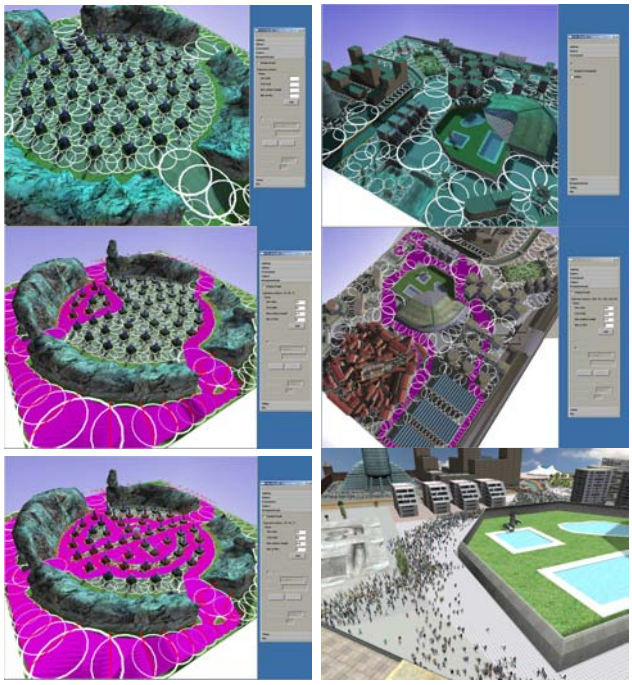

Fig. 3. *left column:* A Stonehenge-like environment and a set of paths solution to a Navigation Flow query with varying maximum relative path length values (Alg. 2). *right column:* A virtual city, a set of paths solution to a Navigation Flow query and a crowd navigating in the city.

the scene creates many congestion points. In such a case, it is important to obtain alternative solutions to a navigation flow query. Indeed, many entities navigating along a single solution path would result in congestions whereas parts of the environment close to the pedestrians would remain empty. This would seem very unrealistic to a spectator. Figure 3 shows solutions to a query for which we have limited the number of solutions (limitation is stronger in the middle image). Without limitation, the union of solution paths covers the whole environment, as shown in the accompanying video. The accelerated part of the video illustrates a real-time simulation of 1000 pedestrians all navigating between identical locations, but dispatched according to the distribution done line 10 of the Algorithm 3. In the virtual city environment, our simulator is able to reach a 35'000 pedestrians crowd with interactive rates (10-20Hz, including rendering tasks, according to the point of view). The obtained performance

is possible thanks to the scalable simulation and rendering: at the forefront, complex articulated characters are rendered whereas in the background simplified representations of humans are used. The crowd is dispatched on the whole city using 7 Navigation Flows of 5'000 pedestrians each, joining main buildings (hotel, church, train station, circus, etc.). Each navigation flow is created in a second (the corresponding $\mathcal{NG}$ is made of 1'500 vertices), which allows interactive crowd setup. The environment geometries are complex: 10'000 and 100'000 triangles approximately for the Stonehenge-like and the City examples respectively. $\mathcal{NG}$ allows to abstract the geometries of the environment and the complexity of both our planning and simulation. Complexity then becomes mainly dependent on the number of vertices and edges composing the graph.

VI. CONCLUSIONS AND FUTURE WORKS

We have presented a method to plan and simulate the navigation of crowds of moving entities in large virtual environments. Our solution is based on a structure called *Navigation Graphs*, which decomposes an environment of any kind in sets of interconnected navigable areas. A specific navigation planning technique allows to dispatch a crowd of moving entities navigating between any pair of given locations. We have also introduced a scalable simulation loop, which allows crowd situation update while distributing the available computational resources in space and time. We have equally been able to preserve quality at its best in the foreground of the central area of the screen as well as real-time rates. Finally, we have presented algorithms to solve useful neighbor and visibility queries. Our method is efficient in the case of large crowds. Indeed, the moving entities' positions are updated block by block, according to their current relative position to the actual observation point of view. We save precious computation time by not accessing each of the entities at each update loop. Our method is demonstrated on crowds of virtual pedestrians with real-time visualization experience, however, its principle is general.

In Robotics, the method is applicable to the simulation of a robot in presence of a crowd of virtual humans (VHs). The simulation of VHs can then be scaled using our method: their behavior is complex enough to simulate interactions with the robot, while VHs in the background only execute a navigation task. Another application for the Robotics field is to adapt the navigation flow algorithm to obtain several solutions to a single query. Thus, additive criteria could be used to select a solution path to a robot navigation query: width of the passages to cross, visibility over given areas along the path, access goal destination from given direction, etc.

Further works are in progress to address dynamic environments, and more specifically to treat the case in which a passage is partially or totally obstructed by a new obstacle (permanently or not) while the simulation runs. In this case, the Navigation Graph must be adapted interactively (deletion, split or addition of vertices and edges) and previously computed paths must be reconfigured (validity checks, new path searches), as well as moving entities' path reconfiguration.

VII. ACKNOWLEDGMENTS

The authors would like to thank Barbara Yersin and Jonathan Maïm for developing the crowd simulator we have used to produce the images in Figure 3 and for their help during experiments, Mireille Clavien and Renaud Krummenacher for environment design, and Mireille again for video production.

REFERENCES

[1] J.-C. Latombe. *Robot Motion Planning*. Boston: Kluwer Academic Publishers, 1991.

[2] Jean-Paul P. Laumond. *Robot Motion Planning and Control*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[3] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[4] Xuejun Sheng. Motion planning for computer animation and virtual reality applications. *CA*, 00:56, 1995.

[5] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90–98, 1986.

[6] Parris K. Egbert and Scott H. Winkler. Collision-free object movement using vector fields. *IEEE Computer Graphics and Applications*, 16(4):18–24, 1996.

[7] Tomas Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, 32(2):108–120, 1983.

[8] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Proceedings of IEEE Transactions on Robotics and Automation*, pages 566–580, 1996.

[9] James Kuffner and Steven LaValle. Rrt-connect : An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation*, 2000.

[10] M.G. Choi, J. Lee, and S.Y. Shin. Planning biped locomotion using motion capture data and probabilistic roadmaps. *SIGGRAPH'03: ACM Transactions on Graphics*, 22(2):182–203, 2003.

[11] Julien Pettré, Jean Paul Laumond, and Thierry Siméon. A 2-stages locomotion planner for digital actors. *SCA'03: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 258–264, 2003.

[12] A. Kamphuis and M.H. Overmars. Finding paths for coherent groups using clearance. *SCA'04: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 19–28, 2004.

[13] M. Sung, L. Kovar, and M. Gleicher. Fast and accurate goal-directed motion synthesis for crowds. *SCA'05: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 291–300, 2005.

[14] Steve Rabin. *AI Game Programming Wisdom*. Charles River Media, Inc., Rockland, MA, USA, 2002.

[15] James Kuffner. Goal-directed navigation for animated characters using real-time path planning and control. *CAPTECH*, pages 171–186, 1998.

[16] R. Bohlin. Path planning in practice; lazy evaluation on a multi-resolution grid. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2001.

[17] W. Shao and D. Terzopoulos. Autonomous pedestrians. *SCA'05: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 19–28, 2005.

[18] F. Lamarche and S. Donikian. Crowds of virtual humans : a new approach for real time navigation in complex and structured environments. *Eurographics'04: Computer Graphics Forum*, 23(3):509–518, September 2004.

[19] S. J. Fortune. Voronoi diagrams and delaunay triangulations. *CRC Handbook of Discrete and Computational Geometry*, pages 377–388, 1997.

[20] Julien Pettré, Pablo de Heras Ciechomski, Jonathan Maïm, Barbara Yersin, Jean-Paul Laumond, and Daniel Thalmann. Real-time navigating crowds: scalable simulation and rendering: Research articles. *Comput. Animat. Virtual Worlds*, 17(3-4):445–455, 2006.

[21] C. W. Reynolds. Steering behaviors for autonomous characters. *Proc. of Game Developers Conference*, pages 763–782, 1999.