Real-Time Multi-Agent Path Planning on Arbitrary Surfaces

Rafael P. Torchelsen¹ Luiz F. Scheidegger¹ Guilherme N. Oliveira¹ Rui Bastos² João L. D. Comba¹
¹Instituto de Informática - PPGC, UFRGS - Brazil
²NVIDIA Corporation

Figure 1: Four different views of a scene where agents navigate on the surface of a complex triangular mesh. Agents are color-coded by their different objectives. The system supports path planning of multiple agents on non-planar surfaces, and imposes no limitation on the domain mesh, such as this mesh with more than one genus.

Abstract

Path planning is an active topic in the literature, and efficient navigation over non-planar surfaces is an open research question. In this work we present a novel technique for navigation of multiple agents over arbitrary triangular domains. The proposed solution uses a fast hierarchical computation of geodesic distances over triangular meshes to allow interactive frame rates, and a GPU-based collision avoidance technique to guide individual agents. Unlike most previous work, the method imposes no limitations on the surface over which the agents are moving, and can naturally deal with non-planar meshes of arbitrary genus and curvature. Moreover, the implementation is a hybrid CPU/GPU algorithm that explores the current trend of increasing the number of CPU cores and GPU programmability. This approach exploits the best qualities in each processor, thus achieving very high performance.


Keywords: Path Planning, Games, CUDA, Geodesic Distances

1 Introduction

Path planning for multiple agents is an open research topic that has been increasingly explored in the literature, especially in the game development community. Issues like navigation and interaction between multiple agents, as well as scalability with respect to scene size and agent number, have been recently addressed to provide increased realism in dynamic and complex scenes. Recent research efforts can be divided into two main approaches: those based on autonomous interaction between agents and those dealing with directing whole crowds to a given destination. Algorithms that address both approaches are, as a rule, computationally expensive and unsuitable for real-time simulations.

Furthermore, most solutions to date limit the movement of agents to the Euclidean plane. This restriction is unsuitable for scenes where agents need to move over curved surfaces. In this work, we propose a method for navigation of multiple individuals on non-planar surfaces, allowing multiple destinations and collision avoidance between agents. Our technique imposes no limitations on the nature of the underlying mesh, which can have any genus (see Figure 1), as well as arbitrary shape. To the best of our knowledge, this is the first work to propose a completely general approach to non-planar path planning.

The main contributions of our work can be summarized as follows:

- A method for multi-agent navigation on non-planar triangular meshes using geodesics
- A multi-resolution technique that allows new destinations for agents to be defined at runtime, while maintaining interactive frame rates
- An efficient GPU-based approach for collision avoidance between agents with minimal synchronization, which maximizes performance by exploring the massively parallel architecture of modern GPUs
- A hybrid CPU/GPU path planning pipeline that explores the best qualities of each processor

2 Related Work

Path planning of multiple agents can be summarized in terms of two main concerns: global and local navigation. Global navigation simply deals with the path an agent must follow to its destination, while local navigation ensures that agents will avoid obstacles and other agents, while minimizing deviation from the original path.
The proposed path planning method computes geodesic distances over a mesh to define the global path, while using a novel local navigation method. This new approach is a parallel algorithm with very few synchronization points, which makes it particularly suitable for GPU implementation. In what follows, we discuss previous work concerning global and local navigation.

2.1 Global Navigation: Distance Computation

Agents moving in a virtual environment typically have to navigate towards a set of objectives. The global path that each agent follows to its goal can be defined by several techniques. The agents can walk directly towards their targets [van den Berg et al. 2008a] or they can move following precomputed roadmaps, as is the case of van den Berg et al. [van den Berg et al. 2008b]. Other approaches for defining the global path of an agent also avoid sharp turns due to obstacles along the way. Thus, the path chosen is a compromise between the shortest possible path and a solution that smoothly dodge static obstacles. This approach is discussed in the work of Silveira et al. [Silveira et al. 2008].

These techniques, however, are restricted to planar domains. More general approaches that can deal with non-planar surfaces usually involve computing distances on the surface of a triangular mesh. This can be done using several algorithms, such as Dijkstra’s algorithm [Dijkstra 1959] (and its variations), Fast Marching (FM) Methods [Sethian 1999] and Window Propagation [Surazhsky et al. 2005]. Dijkstra’s algorithm and the FM methods give approximations of the geodesic distance on a mesh, while the Window Propagation technique gives exact results.

Because of its relatively low cost most applications that require interactivity use Dijkstra’s algorithm to compute approximations to geodesics. However, recent work by Torchelsen et al. [Torchelsen et al. 2009] introduced a new method to compute approximate geodesic distances that is more accurate than Dijkstra’s methods and is still capable of maintaining interactive frame rates. Furthermore, Moreira et al. [Moreira et al. 2005] demonstrate that using Dijkstra’s algorithm to approximate geodesic paths can lead to poor results when compared to FM methods.

2.2 Local Navigation: Collision Avoidance

Collision between agents becomes an important concern when agents are moving in a sufficiently crowded environment. Local navigation techniques exist to avoid collision while minimizing deviation from the globally optimal path. Local navigation methods have been proposed in the literature, as in van den Berg et al. [van den Berg et al. 2008a] and Guy et al. [Guy et al. 2009], who extended the concept of Velocity Objects [Fiorini and Shiller 1998] from robotics to scenarios densely populated with agents. Van den Berg et al. assume similar collision avoidance reasoning for every agent, which results in collision-free and oscillation-free motion.

Fulgenzi et al. [Fulgenzi et al. 2007] apply the concept of Probabilistic Velocity Obstacles (PVO) [Kluge and Prassler 2004] to a dynamic occupancy grid, to estimate the probability of collision when there is uncertainty in the position, shape and velocity of the obstacles. Lamarche and Donikian [Lamarche and Donikian 2004] propose a method for real-time navigation on complex scenarios using topological structuring on the geometry of the environment. Finally, Paris et al. [Paris et al. 2007] present a predictive agent-based approach where the agents react to possible collisions by sampling their surroundings and extrapolating their trajectories.

Some authors also focus on crowd behavior. Yeh et al. [Yeh et al. 2008] introduce the concept of Composite Agents to model complex interaction among individuals. Their technique is based on emergent behavior among agents, such as aggression and priority. Foudil and Noureddine [Foudil and Noureddine 2006] address collision avoidance by introducing priority rules and social behavior to the individuals. Other authors, such as Silveira et al. [Silveira et al. 2008], propose hybrid algorithms that combine global paths and collision avoidance in their techniques.

It is important to note that, in most prior work, local navigation relies heavily on distance measurements between agents and obstacles. This makes such techniques unsuitable for use in non-planar surfaces, where computing distances is typically an expensive operation. Our work addresses this problem and enables a fast and reliable local navigation technique over non-planar surfaces.

In the next section, we discuss how the method performs global navigation. We demonstrate how to compute a hierarchical distance field over the domain mesh, and how the path planning pipeline employs separate, parallel threads to construct this distance field efficiently while maintaining interactive frame-rates.

3 Global Navigation

In our path planning technique, each agent moves toward one of \( n \) different objectives on the mesh. Each of these objectives is defined by a point \( p_i \) over the surface \( S \) of the domain, and for each \( p_i \) we construct a distance field \( f_i : S \rightarrow \mathbb{R} \). As can be seen in Figure 2, given any point on the mesh, the field \( f_i \) determines the shortest distance to the objective \( p_i \). The gradient \( \nabla f_i \) of a distance field gives the direction of the shortest path (also known as the direction of steepest descent) from any point on the domain to its corresponding objective. Therefore it is this direction that agents follow in our method. Using distance field gradients as the direction of the agents’ movement has the advantage that any single individual deviating from the optimal route to avoid obstacles can easily return to the shortest path once it has moved around the obstruction. Also, this technique causes less collisions than roadmaps, since roadmaps tend to group agents close together on the same path.

Most path planning techniques to date have used Dijkstra’s algo-
rithm or its variations to compute distance fields over a mesh due to its low computational overhead. However, distance fields created in this way are very rough approximations of the correct, geodesic distance field, and their paths of steepest descent are not very smooth. Recent work by Torchelsen et al. [Torchelsen et al. 2009] addresses the computational cost of algorithms that compute distance fields over triangular meshes. Their technique constructs a simplified representation of the input mesh, where distance fields can be computed more efficiently. Furthermore, their method allows arbitrary techniques, such as PM methods, Window Propagation or Dijkstra’s algorithm to be used to compute distance fields.

Although the work of Torchelsen et al. improves the performance of distance field computation in general, simultaneously computing many different distance fields over a mesh is still a costly operation. Since this is a requirement for a system that supports multiple goals, we have designed a hybrid CPU/GPU pipeline that can sustain interactive frame-rates by scheduling different threads to gradually compute distance fields in increasing resolutions.

3.1 Hierarchical Computation of Geodesics

The first step in our path planning pipeline involves computing the distance fields over the mesh on the CPU, as can be seen in Figure 3. We use the CPU for this task due to the nature of distance field computation. Distance field algorithms typically employ a variable advancing front over the mesh, which is hard to implement on classical GPU programming models.

Naturally, an agent bound to objective $p_i$ requires that $\nabla f_i$, and hence $f_i$, be computed before it can start moving. In order to maximize interactivity, we use Torchelsen et al.’s technique [Torchelsen et al. 2009] to compute a hierarchy of distance fields for each objective. This computation proceeds from a coarse version of the mesh all the way to the original domain, and is done in parallel threads that do not block the simulation. Thus, agents can start moving almost immediately, and will follow a progressively more accurate path as higher-resolution distance fields become available. Whenever an agent receives a new goal $p_i$, it immediately sends a message to the system requesting $\nabla f_i$. If no version of $\nabla f_i$ is available, this request is added to a queue of distance fields that need to be computed. This queue is always traversed from lowest to highest resolution, to ensure that agents can start moving as soon as possible.

Distance fields and their gradients are transferred to the GPU as soon as they are completed by the CPU threads, and the system tells agents to update their distance fields every time higher resolutions become available for use. Due to a limitation on the architecture of current GPUs, we must preallocate memory for all versions of the distance fields at start-up. This is due to the fact that current GPUs do not allow dynamic memory allocation during code execution.

Although this system efficiently provides smooth shortest-paths for agents to traverse, it does not in any way avoid collisions between agents and obstacles. To address this, we propose a simple grid-based obstacle detection and avoidance approach, which we describe in the next section.

4 Local Navigation

Even though steepest descent lines along a single distance field will never cross, when dealing with multiple objectives in a single scene it is entirely plausible that the path of different agents will intersect, potentially leading to collision between them. We propose a simple grid-based method that deals with collision avoidance while minimizing deviation from the optimal path. In the following sections, we describe the collision avoidance grid and the procedure to determine a new moving direction for agents that must round obstacles.

4.1 Collision Grid

Agents need to know, with some anticipation, whether there are obstacles in their intended path, so they may change their direction and avoid collisions. Our method uses a static 3-dimensional grid to store the current position of each agent in the scene. Each cell in the grid holds a binary flag indicating whether or not there is an agent in that cell. We chose this simple structure because it minimizes the need for synchronization between parallel threads controlling multiple agents, turning it ideal for GPU implementation. Blei-
One drawback of our collision avoidance method is that agents only tag a single cell in the grid. Nevertheless, the grid sampling strategy ensures that agents will be able to detect one another even when they are larger than the cell they occupy. Since agents are controlled by parallel threads, the grid cell tagging must be done as an atomic write. It is, however, the only synchronized operation in our algorithm. Even if agents are larger than the grid cell size they only mark a single cell. This improves performance by minimizing the number of critical sessions involved. Our grid sampling strategy ensures that agents will be able to detect one another even when they are larger than the cell they occupy.

When attempting to move forward agents first sample the grid in a neighborhood of their current position and intended path, searching for tagged cells. If they find any obstacles they randomly choose a small amount of time to stop and then select a new moving direction. The stop is intended to give passage to other agents and reduce the chance of a deadlock. We explain the process of choosing a new direction in further detail in section 4.2. Figure 4 illustrates the collision test process.

In our algorithm, local navigation is affected by three main parameters: the size of the sampling neighborhood, the grid resolution and the discrete step $r$ taken by agents for each time step. The radius of the sampling neighborhood is defined by the ratio between the agent’s radius and the grid resolution. Thus, if the agent is much larger than a grid cell, it will accordingly check more cells around its neighborhood. This ensures that agents will not collide even though they only tag a single cell in the grid. Nevertheless, the grid resolution should be close to the agent size, striking a balance between memory usage and accuracy of the results. Finally, $r$ is a discretization parameter that is usually set as the radius of the agent.

One drawback of our collision avoidance method is that agents only avoid one another when they are relatively close. A more refined approach could be achieved by making agents modify their distance fields, for example by adding a Gaussian bump to their local neighborhood. This would make other agents smoothly avoid contact, but at an increased cost, since the fields would have to be modified dynamically. This would leave few computational resources available for other parts of an interactive application, such as rendering and, in the context of games, AI and sound. Our approach is a simple compromise with small computational overhead.

Another advantage of using a 3-dimensional grid, as opposed to a structure defined directly on $S$, is the ability to naturally support external obstacles not on the mesh. Moreover, two agents can be close enough to collide even though the distance between them on $S$ might be arbitrarily large. Figure 5 depicts an example of these cases. Only a true 3-dimensional data structure can naturally deal with these situations.

### 4.2 Finding an Unobstructed Direction

In order to simplify our collision avoidance scheme, the result of sampling the grid is a binary answer. Agents have no knowledge as to where along their path the collision occurred. Therefore, our collision avoidance mechanism searches for new paths around the original one, and repeats the grid sampling in these new directions.

The modified path through which an agent will move is defined by following a new vector constructed by rotating $\nabla f_i$. This makes the agent follow a logarithmic spiral that converges on the objective, as we illustrate in Figure 6. By traveling on this spiral, instead of the original path, agents can smoothly avoid obstacles along the way.

To determine the rotation angle for the spiral, agents progressively test wider rotations around the original path, as depicted in Figure 7. For each candidate rotation, the agent repeats the collision test described earlier. This process terminates either when the agent finds a free path, or when the rotation angle reaches a user-defined upper limit. This upper limit, $\beta$, exists to ensure that agents will not curve too sharply. If the agent cannot find a rotation smaller than $\beta$, it stops for a small random amount of time, waiting for its neighboring agents to move out of the way. Note that a small $\beta$ may result in a agent holding forever in front of a static obstacle. However, if the obstacle is static it can be considered in the gradient generation, that way, the global navigation would give a collision free route. This can be done by removing, from the surface mesh, the intersection of the obstacle with the surface.
The agents gradually blend the rotated vector with the original direction. This results in a smooth transition between the path of steepest descent and the collision-aware route. A drawback of this approach is that some collisions may still occur in very specific conditions where agents can’t dodge one another quickly enough. In our experiments, however, this was not the dominant situation.

After an agent has rounded an obstacle, it can safely return to the path of steepest descent (see Figure 8). Therefore, we reset the rotation angle and repeat the collision test at every time step.

5 Results

We have conducted our experiments on an Intel Core 2 Quad running at 2.83GHz, with 3GB of RAM, and an NVIDIA GeForce GTX260 graphics card. We have tested our system with three different meshes, the Stanford Bunny, the Fertile dataset and the Moai Statue. Table 1 shows the size of these datasets.

As can be seen in Figure 9, our method performs at interactive frame rates even with a large number of agents. It is interesting to observe that the performance of the system is practically constant when the number of agents is less than 512. This is due to the fact that, as long as this number is less than the amount of threads supported in the graphics card, all the work is parallelizable. Once the number of agents goes beyond this value, the card must make more than one serial pass to generate the results. Therefore, it is expected that the performance will decrease gradually as the number of agents increases by increments of 512. Also, we have included a graph showing the performance without rendering, to better highlight the computational cost of our path planning technique.

Although Figure 9 shows results for only a single mesh (the Fertility model), we have observed that all meshes have a similar perfor-
Figure 9: Performance Graphs. Depicted here are performance results of our algorithm. We measured FPS in an interactive system while varying the number of agents and the resolution of the collision grid. On the left, we illustrate the results of running our path planning algorithm with rendering enabled, while on the right we show the performance of the path planning pipeline running without rendering.

Table 1: Test Datasets. This table shows the size of the datasets we used as surfaces for our path planning algorithm.

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Vertex Count</th>
<th>Face Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fertility</td>
<td>4994</td>
<td>10000</td>
</tr>
<tr>
<td>Moai Statue</td>
<td>10092</td>
<td>20180</td>
</tr>
<tr>
<td>Stanford Bunny</td>
<td>34834</td>
<td>69664</td>
</tr>
</tbody>
</table>

It is interesting to note that the performance of our system does not depend on the size of the datasets. This is due to the fact that our implementation allows agents to fetch $\nabla f$ in constant time. Also, the performance loss that we observe when increasing the resolution of the grid is probably due to caching behavior on the graphics card, since the grid is stored in random access memory.

Figures 10 and 11 show our path planning algorithm running on two different datasets. In both cases, the agents are color-coded by their objective, which is also rendered as a solid sphere.

6 Discussion and Limitations

Our global navigation approach is similar to existing techniques in the literature, where we define agent movement by the gradient of a distance field. However, we found no existing mechanism of local navigation that could easily be extended to arbitrary surfaces. Therefore, we designed a simple grid-based collision avoidance system that strikes a compromise between collision-free navigation and smooth movement. While our solution does not, in principle, guarantee that no collisions will occur, we have conducted experiments and concluded that collisions are avoided in most of the situations, which is acceptable for games given the ability to navigate on arbitrary meshes. One of the main advantages of tagging only one cell of the grid is the reduced concurrent writing. However, it assumes a sphere or cube bounding box around the agent. For handling less-regular-shaped agents, we could store shape information, along with the tagging in the collision grid. The collision test would also have to be enhanced for using irregular shapes.

We have designed and implemented a solution meant to be practical. Although the 3-dimensional grid has limitations, such as forcing a compromise between memory usage and simulation accuracy, the simplicity of this data structure makes it very well suited for implementation on the GPU. The massively parallel nature of this processor is very attractive to regular data structures, which need no dynamic updates and have a predictable access pattern. This access pattern naturally improves the performance of our algorithm, due to efficient memory caches. Also, our technique imposes an affordable computational overhead, and can therefore be easily integrated into larger applications, such as games and VR environments.

Figure 5 depicts a situation where two agents that are close together will collide even though they are far apart on the surface. This is correct and expected behavior, but our solution might fail in the opposite case, that is, when two agents are on opposite sides of a positively curved surface, such as a sharp bump on the mesh. This can happen mainly due to an excessively coarse grid, resulting in an overly conservative collision avoidance test.

Our approach aimed at avoiding collisions without trying to mimic particular behavior of the agents (ants, humans, etc.). The motivation for our approach was performance, which usually limits the design of most path planning algorithms in games. In general, games use the same path planning algorithm for all kinds of agents, only changing their size and speed, due to the computational cost for handling more complex agents.

7 Conclusion and Future Works

In this paper, we have presented a novel approach for multi-agent path planning on arbitrary surfaces. To our knowledge, this is the first work in the literature to address this problem. Our solution makes no assumptions about the domain mesh, and can handle surfaces of arbitrary shape and genus. We use a hierarchical computation of geodesic distances on the domain to define a scalar field
whose gradient smoothly guides the agents to their goals. In order to avoid collision between agents we designed a grid-based local navigation algorithm that can also handle obstacles not on the mesh. Our collision avoidance technique is approximate, but we have conducted many experiments and found that few collisions occur. We have also integrated these techniques into a hybrid CPU/GPU pipeline carefully designed to exploit both processors particular strengths. We have implemented the hierarchical distance field computation on the CPU, and the massively parallel collision avoidance algorithm entirely on the GPU, using CUDA.

Currently, our method requires that the domain mesh be static. To allow dynamic domains it would be necessary to modify the hierarchical distance field computation to update its results every time the mesh changes, as well as update the agents’ position to ensure that they are still properly on the mesh. Another challenge for future work is to design and implement a more refined data structure to deal with collision avoidance. Hierarchical structures such as kD-trees or Octrees present themselves as attractive alternatives. Another direction of research is to compute local navigation in the parametric domain, using the work of Torchelsen et al. [Torchelsen et al. 2009]. This would allow the use of existing local navigation techniques, because the parameterization is defined in 2D. How-
Figure 11: The Moai Statue. Time step of the algorithm viewed from two different positions, simulating 512 agents on top of the Moai Statue. In this image we can observe that some collision cases occur, as can be seen on the nose of the model. This is rare, however, and happened in this case only because there are two objectives very close together (the green and blue-gray objectives). Ever, the local navigation would only consider collisions on the surface of the mesh and performance might become an issue for real-time applications.

8 Acknowledgments

We would like to acknowledge NVIDIA for equipment donations. This work is sponsored by CNPq-Brazil grants: 140241/2007-8, 114136/2009-2, 132570/2009-2, 485853/2007-8 and 569239/2008-7. The Moai Statue and Fertility meshes were provided by the AIM@SHAPE project (www.aimatshape.net) and the Stanford Bunny was provided by the Stanford Computer Graphics Lab.

References


