Real-Time Local Unfolding for Agents Navigation on Arbitrary Surfaces

Iago U. Berndt, Anderson Maciel
Instituto de Informtica (INF)
Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil

Rafael P. Torchelsen
Centro de Desenvolvimento Tecnolgico (CDTec)
Federal University of Pelotas (UFPel)
Pelotas, Brazil

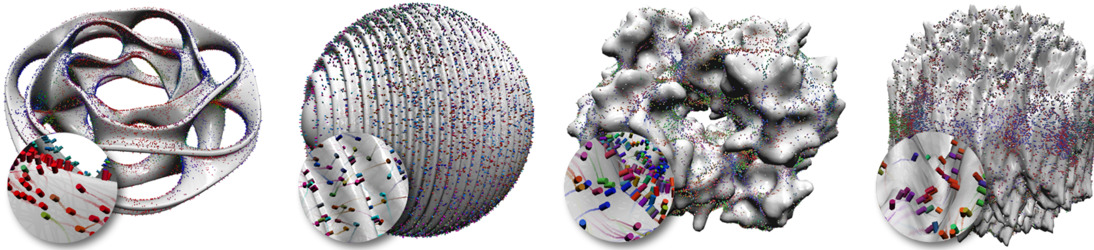


Figure 1: Examples of the meshes used in our tests. From left to right: Heptoroid, Waves, Glog and Meteorite. Our method can handle 140,000 agents in real-time with current consumer class hardware. We used only 20,000 agents in these images to allow visual inspection.

***Abstract*—Agents path planning is an essential part of games and crowd simulations. In those contexts they are usually restricted to planar surfaces due to the huge computational cost of mapping arbitrary surfaces to a plane without distortions. Mapping is required to benefit from the lower computational cost of distance calculations on a plane (Euclidean distance) when compared to distances on arbitrary surfaces (Geodesic distance). Although solutions have been presented, none have properly handled non-planar surfaces around the agent. In this paper we present mesh parametrization techniques to unfold the region around the agent allowing to extend to arbitrary surfaces the use of existing path planning algorithms initially designed only for planar surfaces. To mitigate the high computational cost of unfolding the entire surface dynamically, we propose pre-processing stages and massive parallelization, resulting in performances similar to that of using a planar surface. We also present a GPU implementation schema that permits a solution to be computed in real-time allowing agents to navigate on deformable surfaces that require dynamic unfolding of the surface. We present results with over 100k agents to prove the approach practicality.**

***Keywords*-path planning; agents; computer graphics;**

I. Introduction

Path planning restricted to a 3D mesh surface is an emerging research topic. The applicability to several fields is one of the motivations. Scenes where several agents move as a crowd on an irregular surface have become familiar in movies. On the other hand, for games and real-time crowd simulations their use is still limited. One of the main reasons is the lack of a technique that integrates the existing path planning, mostly designed for planar surfaces, and arbitrary meshes with a low computational cost. This is an important aspect due to the considerable number of path planning algorithms in the literature. Although those can be used on non-planar surfaces the computational cost is prohibitive. For example, most methods use distance computation between agents, which on planar surfaces is a *cheap* Euclidean distance. On the other hand, on irregular surfaces a geodesic distance is necessary which presents a considerably higher computational cost.

Recent works have presented solutions that avoid the higher costs of geodesic distance computation during local obstacle avoidance. Those methods assume that the agent is always surrounded by a planar surface, allowing the use of path planning techniques designed for planar surfaces on an arbitrary surface. Fig. 2 illustrates the problem of assuming that the region around an agent is planar on an arbitrary surface. At each frame each agent must compute a new direction and speed according to their surroundings. To avoid using geodesic distance computation the usual approach is to orthogonally project the triangles around the agent onto the plane of the triangle where the agent is located. This results in all neighboring agents to be mapped to the same plane where Euclidean distance can be computed. Notice, however, that several collateral problems occur. Agents relatively distant may be mapped very close to each other. In addition, the heading direction of each agent is also mapped, which causes navigation errors, as illustrated in Fig. 2. On the other hand,

unfolding the mesh around the agent avoids all those problems.

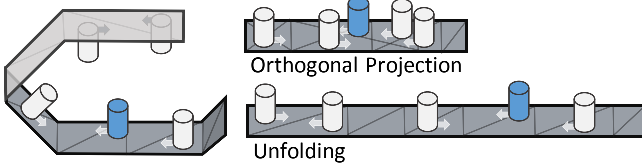


Figure 2: The left mesh illustrates an arbitrary surface with several agents moving. The orthogonal projection is the approach used in previous works. It causes triangle overlap and inconsistencies on the agent's heading directions. Instead, our proposal is to unfold the mesh around the agent, resulting in the bottom right situation that is suitable to any planar navigation algorithm.

To mitigate the distortion and allow the use of existing path planning techniques on arbitrary meshes our proposal is to unfold the region around each agent using a mesh parameterization technique. This is similar to what advanced texturing techniques do. To demonstrate the benefits of this approach, we present results of agents navigating on animated meshes. In addition, for static meshes, we propose a pre-processing stage that avoids the computational cost of dynamically unfolding the mesh. Results in Sec. IV show a smaller computational footprint compared to using a planar region during the obstacle avoidance stage, allowing the use of very complex meshes with more than 100,000 agents in real-time. Different from similar works, we show results that explore GPU parallelism, also presenting a practical solution for real-time applications.

II. Related Work

Path planning for agents or crowds is an active research topic. However, most of the previous works are focused on path planning for planar surfaces [1], [2], [3], [4], [5]. Although planar surfaces are the most common surfaces where agents navigate there are applications, especially games, where the agents are uncommon and walk over any surface. Movies and simulations of insects, for example, also require a method to use path planning on arbitrary surfaces.

To allow unrestricted agents movement on a 3D mesh surface the path planning technique must consider limitations that are not present in planar surfaces. The first work to explore this topic used simple grid subdivision of the space to identify collision routes [6]. Although the method can compute paths on arbitrary meshes for several agents, the collision avoidance was prone to errors. Another work [7] used a collision avoidance method [8] on non-planar surfaces. However, navigation was only on a simplified version of the mesh that is locally planar. Assuming that the mesh is always planar around the agent allows the use of existing path planning algorithms. However, this imposes a limitation to the complexity of the mesh that must be almost entirely planar.

Jund et al. [9] presented a unified structure to account for path planning and proximity queries. To be able to use RVO2 algorithm [2] for local collision avoidance they flattened the region around the agents. The paper is not specific about the method used to flatten the mesh, but the surfaces used and the agents size result in quasi-planar regions around the agent, where the Euclidean distance and geodesic distance would be very similar. The authors present results with scenes that change over time, but those changes are only regarding moving obstacles. In contrast, our approach allows surface animation.

Ricks and Egbert [10] also present a technique capable of computing paths on 3D meshes. Firstly, the optimal path is computed avoiding sharp turns. Then, obstacle avoidance is conducted during the agent movement on the optimal path. Also, collision with obstacles external to the surface are accounted. Similarly to other previous works, the surface around the agent is considered planar and no efficient parallel implementation is proposed.

Another drawback of assuming that the region around the agent is always planar is that it reduces the agent's field of view. For example, Golas et al. [11] proposed a long-range collision avoidance that, if applied to a highly curved mesh using the orthogonal projection, would result in considerable error in the distance computation (Euclidian vs. Geodesic).

Another trend for obstacle avoidance is to mimic the real vision [12]. Such approach would provide limited vision over an intricate mesh when mimicking the agent's vision, which may be desired. The same desired feature can be obtained with ordinary vision techniques, however, using our unfolding approach with a long-range geodesic view.

In face of the exposed limitations of previous works, in this paper we focus on an approach to unfold the mesh around the agent. This accounts for surface animations and allows the use of existing flat surface path planning techniques. Contrarily to previous works, we also present a highly parallel and efficient implementation.

III. Arbitrary Surfaces Unfolding for Agents Navigation

Agent navigation usually involves two stages. The first stage is path planning, where an optimal path is determined, e.g. the shortest path between two points on a surface. The second stage is local obstacle avoidance, where the optimal path is dynamically modified to cope with the influence of moving obstacles. Several methods are already available that can be used in each of the two stages.

Our proposal is to provide an intermediary stage between the other two as a common ground where the navigation is computed. While common approaches for local obstacle avoidance rely on the planification of the terrain around the agent by orthogonal projection, our approach is to locally unfold the surface around each agent. The unfolding is based on traditional texturing techniques and avoids a number of issues

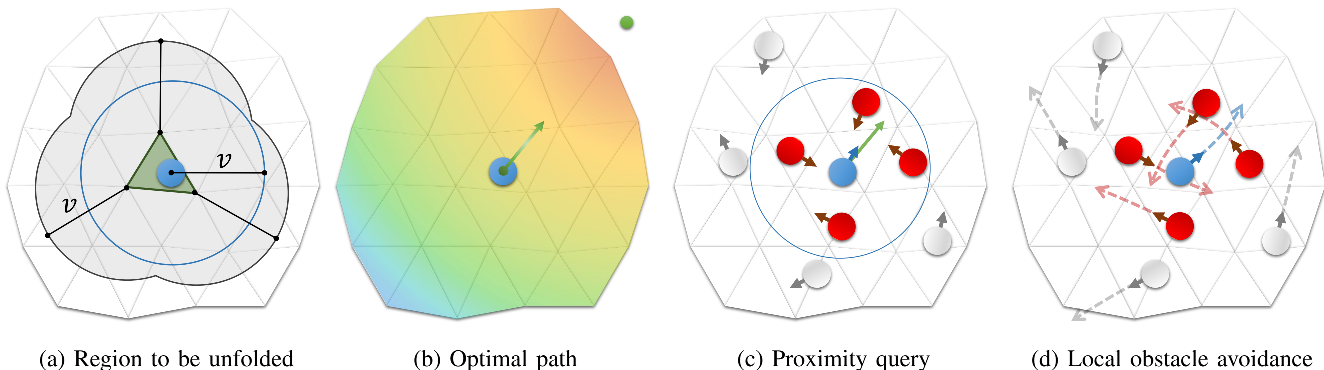


Figure 3: (a) Firstly, a region around a triangle is defined and unfolded in to a chart. (b) The optimal path is computed on the entire mesh and encoded as a distance field; here only the distance field on the chart is visible. The green dot denotes the agent destination. (c) To select the possibly colliding agents, a proximity query is done. (d) Finally, the local obstacle avoidance is computed on the chart plane and the new position is mapped back to the 3D surface.

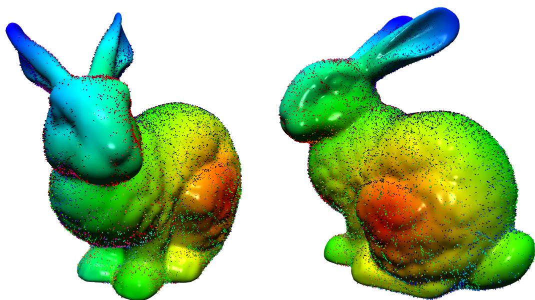


Figure 4: The optimal path encoded as a distance field using the Dijkstra's algorithm. The red pixels are closer to the agent's goal, while the blue ones are the most distant. This example illustrates only one of many distance fields in use.

of the previous methods. It creates a planar representation (chart) where a suitable path can be planned. Fig 3 depicts an overview of this processes.

In the remainder of this section we detail the two traditional stages and the new stage of local unfolding, including a pseudo-algorithm of our method.

A. Optimal Path

The optimal path can be defined as the path that would be followed by an agent when no dynamic obstacle is present in the simulation. Many times it is the shortest path, and it is also called global path. Being optimal depends, however, on the level of accuracy required by the application. Distance over the graph (Dijkstra's, A*) [13] or geodesic distance [14] [15] are commonly used as global paths. Other existing approaches, instead, try to mimic the human behavior [16] [17].

Independently of the method used to build the path, it can be mapped to the surface as the distance field as in Fig 4.

Implementation

In our implementation, the Dijkstra's algorithm is used to estimate the shortest path to the destination. The accuracy is dependent on the mesh tessellation, but it is still a usual and plausible approximation.

We compute the optimal path on the fly using an FIFO queue to minimize simulation stall. In this queue schema, the number of paths computed per frame is given by a time budget. Each path can be computed independently, allowing a multi-thread implementation. We use a CPU implementation where each thread computes one path. Similar implementations exist using the GPU [13]. However, we have opted by the CPU to save the GPU for the unfolding in the next stage (Sec. III-B).

The FIFO approach is efficient especially when the number of agents is smaller than the number of vertices on the mesh surface. However, if the number of agents is greater than the number of vertices, precomputation is more suitable. It is also more memory efficient as many agents will share the same destination points and thus the same map. Precomputing reduces the computational cost during run-time and, even if the mesh deforms, only the deformed regions will require update, which is done with the FIFO approach.

As we intended to simulate a huge number of agents, we pre-processed all possible distance fields, from each vertex, avoiding line 7 of Algorithm 1. The resulting memory complexity is $vertices^2$. Although this results in a considerable memory footprint, the solution is viable for real-time applications, especially games, where the path planning is just one of the several algorithms running.

B. Local Unfolding

To compute a plausible path for each agent on an arbitrary mesh, obstacle avoidance methods require a planar region to be computed. As already stated, we propose to do so by unfolding the mesh instead of projecting it on a plane. Common unfolding approaches are to unfold the entire mesh or non-intersecting regions (clusters) [18]. In any case, it is desirable that the almost inevitable distortions resulting from

the unfolding be away from the agent. The unfolding result is a chart (planar sub-mesh) that is independent of the agent's position on the central triangle of the chart, allowing several agents to populate the same region.

Our method consists in creating one chart for each triangle of the mesh and use it only for the agents on that triangle. Fig 3a show the chart region. The agent's view radius (v) from the vertices of the central triangle determines the chart radius. All triangles with at least one vertex inside this radius are included. The Dijkstra's algorithm is used to walk on the mesh from the central triangle up to the radius limit.

After the chart's triangles are selected that region is unfolded using the LSCM algorithm [19]. We map the central triangle to the plane without distortion, which can be done with a single triangle by an orthogonal projection. We also lock it so the LSCM algorithm do not change it during the unfolding of the remaining of the chart. This is done to avoid distortion on the region where the agent using the chart is located. The unfolding begins by an orthogonal projection of the mesh over a plane as an initial solution, in other words, assuming the surface can be approximated by a plane, similar to previous works [9][10]. However, the initial solution is optimized to minimize distortion and overlapping triangles by the LSCM algorithm [19] which is done by an iterative deformation of the triangles (2D) similar to a mass-spring system [20].

Other unfolding methods exist and could be used as an alternative to LSCM. We refer the reader to references [21], [20] and [22]. In most cases, there will be some distortion that will affect the agent's movement. It is, nevertheless, lower than assuming that the surrounding of the agent is planar as in previous path planning methods. The effect in the agent's movement is explored in the results section.

Implementation

Unfolding techniques often imply pre-processing due to the high computational cost. However, in our implementation the regions that are unfolded contain few triangles. In our tests, the charts contain from 13 to 133 triangles. This causes the cost to process one chart to be small. Moreover, our unfolding defines one independent chart for each mesh triangle, resulting in a highly parallel distribution of work. For this reason, the performance of a GPU implementation allows real-time update of the charts during animation. Lines 1-4 in Algorithm 1 refer to the pre-processing step, while lines 11-13 correspond to the update. Fig 5 show a surface that is being interactively deformed over time. Notice that only charts containing triangles affected by the animation need an update.

C. Local Obstacle Avoidance

Local obstacle avoidance is any algorithm that given an optimal path tries to follow the path while avoiding obstacles. As mentioned before, there are several methods in the literature that solve this problem for planar regions. Our approach allows using any of those methods without modifications with non-planar arbitrary surfaces. Only the input changes to include the unfolded chart (Sec. III-B).

Algorithm 1: Simulation

```
1  for t ∈ triangles do
2      select triangles around t;
3      charts[t] ← unfolding of the region around t;
4  end
5  while True do
6      for a ∈ agents waiting destination do
7          if current frame time budget allow then
8              Dijkstra(a_destination);
9          end
10     end
11     for t ∈ modified mesh region do
12         charts[t] ← unfolded charts[t];
13     end
14     for a ∈ agents do
15         c ← current a chart;
16         RVO2 ← position (2D) of a in c ;
17         RVO2 ← velocity of a in c ;
18         RVO2 ← preferred velocity of a;
19         for i ∈ other agents on c and in a view range do
20             RVO2 ← position of i in c;
21             RVO2 ← velocity of i in c;
22         end
23         new velocity of a ← RVO2;
24         new position of a ← RVO2;
25     end
26     for a ∈ agents do
27         velocity of a ← new velocity of a;
28         position of a ← new position of a;
29         update triangle beneath a (chart);
30         calculate barycentric coordinates of a;
31         if a changed chart then
32             update preferred direction (optimal path) of a;
33         end
34     end
35 end
```

Given the chart we compute the agent's barycentric coordinates on the triangle where it is currently located, creating a mapping between the chart domain and the original surface mesh. The lines in Algorithm 1 containing "RVO2" indicate the interface with the local obstacle avoidance. Three inputs are given to the RVO2 algorithm for the agent and its neighbors: position, velocity and preferred velocity.

Implementation

To compute the direction of the preferred velocity, we determine a reference point that represents an intermediary target for the agent on the chart plane at the same distance as the actual destination (green dot in Fig 3). The actual destination cannot be used as it might cause convergence to a local minimum. The reference point is then calculated, in

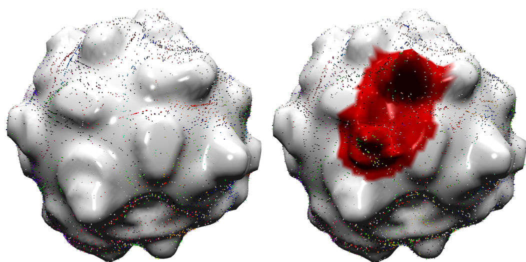


Figure 5: Before and after deformation of the mesh. The red region denotes the charts that are gradually updated (unfolded) while agents navigate.

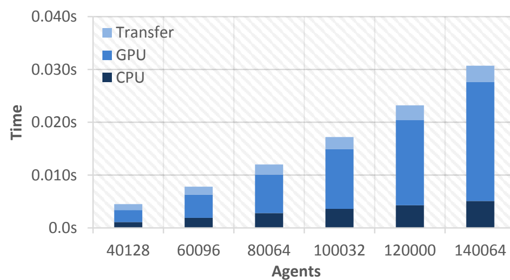


Figure 6: Total computational time of one iteration (Algorithm 1) on the Glob mesh. The time that the CPU is waiting for the GPU is accounted in the GPU processing time.

our implementation, as the sum of the vectors departing from the center of the current triangle to the n vertices around it (Dijkstra's is used again) weighted by the distance of each vertex to the final destination as in Eq 1. We used $n = 10$.

$$refPt = \overline{\sum_{i=1}^{n} dir_i(dist_{center} - dist_i)dist_{center}} \quad (1)$$

Next step is to select agents in the view range of the current agent (lines 19-22 in Algorithm 1). This requires a proximity query (Fig. 3c). Our approach is similar to the one in reference [9]. Each agent is associated to the triangle it is located and, according to the view range of each agent, the surrounding triangles are queried. Only the triangles in the chart need to be queried, as no agent can see beyond it (Fig. 3a). To avoid querying the entire chart we store a distance field from the central triangle to the borders, which also allow agents with different view ranges.

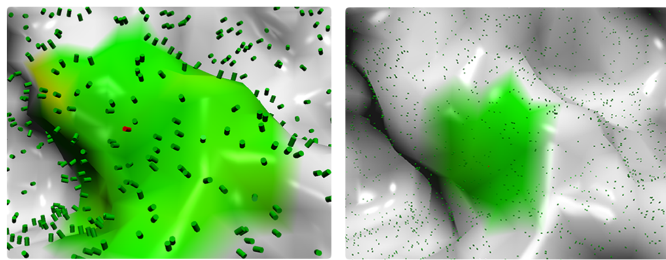
Finally, RVO2 or any other planar navigation algorithm is applied to produce the new velocity and position of the agent (lines 23-24 in Algorithm 1).

IV. Results

Our implementation uses C++, OpenGL, OpenMP, NVIDIA CUDA™, RVO2 Library [23] and OpenNL[24] (unfolding). The following performance numbers have being collected on an Intel I7™4770k, 12GB ram and a NVIDIA GeForce GTX 980™with Microsoft Windows 7™.

The rendering quality used to represent the agents are application dependent. To avoid mixing computational costs, the performance numbers do not consider the rendering cost. We use pre-processing for charts and optimal path on all static models, and we gradually update both on animated meshes.

The charts and local obstacle avoidance are computed in the GPU while the optimal path uses the CPU. Such work distribution allows both processors to work in parallel, although there is a computational cost of transferring the data. Also, most of the time the CPU is waiting for the GPU, due to the higher cost of the local obstacle avoidance with several agents. Fig 6 illustrates the workload distribution.



(a) Surface scale 1x

(b) Surface scale 5x

Figure 7: Two surface scales used on the tests. The right figure is the result of scaling up the surface 5 times. The green region represents the chart (Fig 3a) for the red agent. Note that the agent's size and field-of-view are relatively smaller as the surface size increases, possibly impacting the quality and performance of the path planning.

To demonstrate the method on different situations, the results presented below use two different scales for the surface where the agents navigate. Increasing the surface scale results in a reduced field-of-view (chart) for the agents. It can then be interpreted as a scale-down on the agent size. Notice on Fig. 7 the effect on the number of neighboring agents and on the complexity of the mesh that is unfolded.

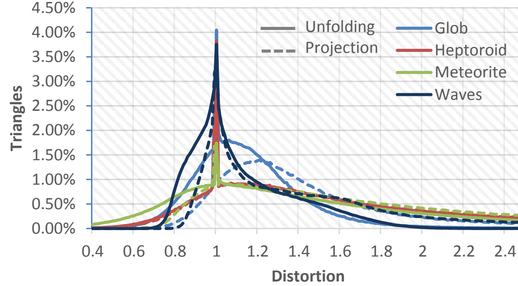
Below, we analyze our results comparing our method (unfolding) with the orthogonal projection of the agents surroundings onto the plane, which is used in previous works. We do this following a number of criteria.

First criterion is distortion. Fig 8 illustrates the L2 stretch [25] measured per triangle per chart. Each triangle is accounted several times because each one is present in several charts. The distortion around the agent is usually lower than near the chart border due to the locking of the central triangle during unfolding. This is not the case with the orthogonal projection method. Overlapped triangles resulting from the orthogonal projection (see Fig 2) also affect the quality of the path planning and is an error that our method avoids. Fig. 9 illustrates the difference in quality.

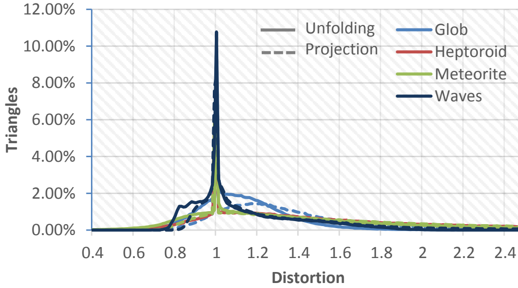
The next criterion is average agent travel time and distance. We tested each surface with increasing number of agents.

Model	Triangles	Time			Memory			Average triangles per chart	
		Chart unfold	Chart unfold (S.5x)	Optimal path	Charts	Charts (S.5x)	Paths	Surf. 1x	Surf. 5x
Glob	29774	0.0579ms	0.0164ms	1.8766ms	28.63MB	10.67MB	0.82GB	36.01	13.43
Meteorite	81920	0.2337ms	0.0143ms	5.2388ms	208.54MB	28.52MB	6.25GB	95.33	13.04
Waves	81920	0.282ms	0.0141ms	4.8174ms	248.62MB	28.44MB	6.25GB	113.65	13.00
Heptoroid	40084	0.0755ms	0.0164ms	2.8737ms	44.53MB	14.56MB	1.49GB	41.60	13.60

Table I: This table shows the computational cost of the method. The unfold time is the average required to unfold a chart. The optimal path is the average time for a single distance field. The memory consumption for the charts increases significantly with the number of triangles. However, this can be computed gradually by prioritizing regions of the surface with agents.

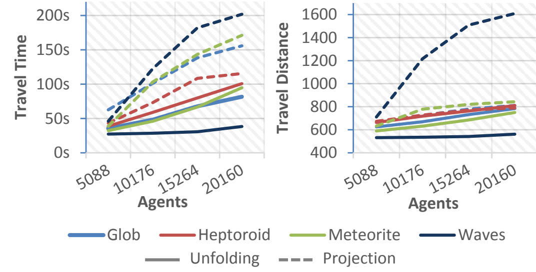


(a) Surface scale 1x

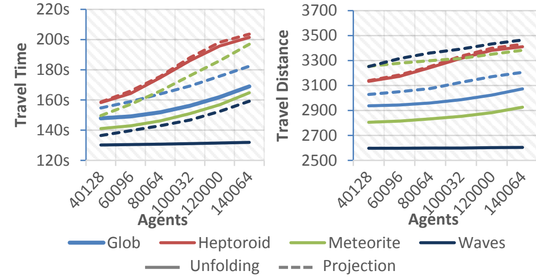


(b) Surface scale 5x

Figure 8: Percentage of triangles according to the L2 stretch. A distortion of 1 means the triangle was not distorted. Notice on Table I that the larger surface result in charts with fewer triangles, decreasing the distortion on both methods.



(a) Surface scale 1x



(b) Surface scale 5x

Figure 9: The average time and distance required for all the agents to reach their goals increases almost linearly with the increase in the number of agents. Notice that orthogonal projection incurs in higher times and distances due to the distortion illustrated in Figs. 2 and 8.

Initially, each agent was randomly placed on the surface. Then, a goal for each agent is randomly computed. To avoid a goal too close to the agent we randomly compute a new one if the travel distance is shorter than the average of the initial goals, and we repeat the processes if necessary. Fig. 9 shows the results. Notice that the travel time and travel distance are lower with our method (solid lines), which means the agents found less errant paths. Also notice that the difference is more significant with the surface in 1x scale, where the unfolded charts are larger and contain more triangles. In 5x scale, the neighboring surface becomes flatter in relation to the size of the agent. There, orthogonal projection (dashed lines) suffers less from distortions and overlap.

In Fig. 9, for the sake of evaluating the path quality, only the agents that reached the destination within a given time limit were accounted. Fig. 10 shows that using projection (dashed lines) many agents get lost, and that this is more significant as the total number of agents increases. All tests have used the same amount of time. With projection, more than half of the agents are still navigating for some surfaces when 20,000 agents are used. The trend indicates that it will be even worse if the number of agents increases. Our unfolding method, on the other hand, finds paths for all agents within the time given for all simulations, being very effective.

Figs. 11 and 12 illustrate the efficiency of our method. Fig. 11 presents the average agent speed, which is affected by the increasing number of agents. Speed using our method is always closer to the desired speed (100%) when compared with projection. Fig. 12 shows the average deviation from the optimal direction (given by the optimal path). Notice that the local obstacle avoidance method has to apply more turns as the number of agents increase, and that the final path is closer to the optimal when the unfolding is used instead of projection.

As for computation efficiency, Fig. 13 depicts the framerate

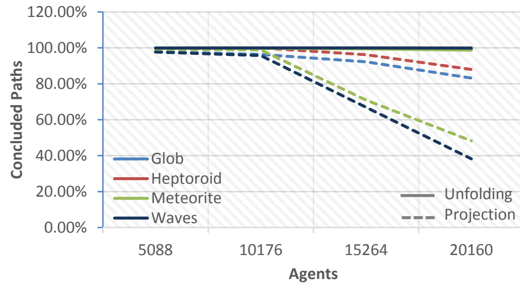
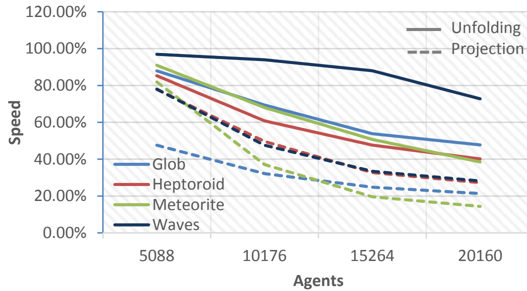
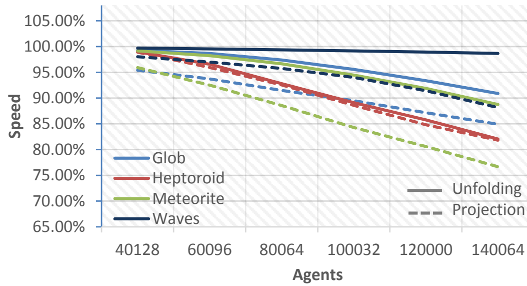


Figure 10: Percentage of agents that concluded their paths in a given time limit.



(a) Surface scale 1x

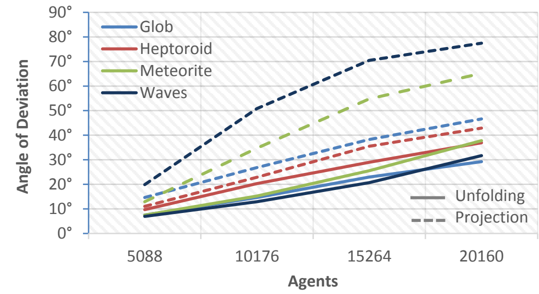


(b) Surface scale 5x

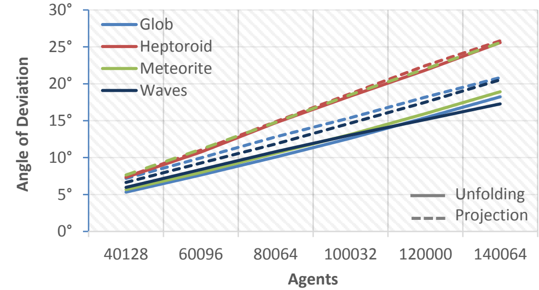
Figure 11: Agents average speed in relation to the desired speed (100%). The speed variation is lower on the larger surface due to the fewer encounters between agents, allowing the agents to follow the optimal path with minimal deviation.

with an increasing number of agents. Notice that the FPS maintains near or above 30 for all meshes even with the unprecedentedly high number of 140 thousand agents.

The depression around sec. 60 in Fig. 13 is caused by a peak in the number of neighbors around that time, as shown in Fig. 14. Table I presents information about the meshes used and the computational cost for pre-processing the charts and optimal paths. Also, it shows the average number of triangles per chart. For surfaces with animation, the performance is degraded by the number of charts and optimal paths recomputed per frame. Notice that both can be recomputed gradually to avoid instability in the FPS. However this results in paths less accurate before the update completes.



(a) Surface scale 1x



(b) Surface scale 5x

Figure 12: Deviation from the optimal path (0°) as the number of agents increase.

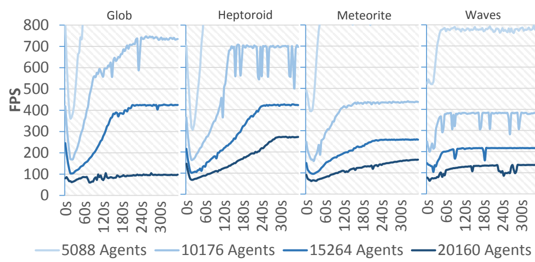
V. Discussion and Limitations

The charts can contain holes, which we do not consider as obstacles. Often, the holes are formed on the base of surface spikes and the chart will surround it. One approach to avoid holes is to cut the chart from the farthest point in the hole border to the closest point in the chart border. We did not pursue this because the hole is usually near the border of the chart due to the chart being formed as a growing region away from the agent position. Also, there was not a significant increase in collision to justify the performance penalty to handle it. Notice that every time the agent crosses a triangle border a new chart is selected, consequently moving the hole away from the agent. The same problem is present when orthogonal projection is used.

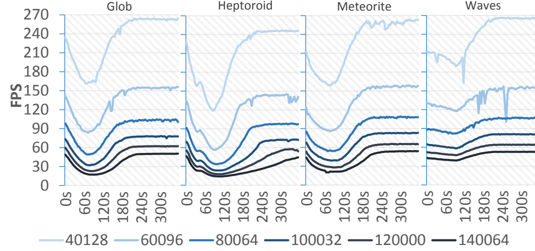
Rendering crowds as the ones in Fig. 1 results in a substantial computational cost. Although our method can compute the paths in real-time for more than 140,000 agents, rendering in real-time is challenging. We did not pursue rendering optimizations (LOD or geometry instancing) in this paper.

VI. Conclusion and Future Works

In this work, we introduced a novel method to allow the use of existing ordinary flat surfaces path planning techniques on arbitrary irregular 3D surfaces. The use of those techniques on arbitrary surfaces was limited, specially in terms of quality of the final paths and low performance. Our results show that the insertion of the unfolding stage between the optimal path and the local obstacle avoidance stages allows real-time



(a) Surface scale 1x



(b) Surface scale 5x

Figure 13: FPS variation during several simulations. Notice the correlation with the number of neighbors in Fig. 14.

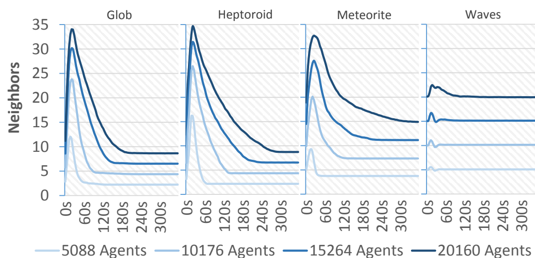


Figure 14: Average variation on the number of neighbors (agents inside another agent view range). The higher the number, the higher the computational cost to avoid collision.

performance for over 100,000 agents, which is more than 10 times higher than previous works [6], [9], [10].

We also proposed a combination of pre-processing stages and update queues that reach performances similar to those previously obtained only on simpler planar surfaces. Thus, our method can handle interactively animated deformable terrains in real-time. We finally analyzed the performance of a massive parallel implementation to prove the method applicability. None of these aspects were present in previous works.

As future work, we would like to explore a levels-of-detail approach to the path planning of agents occluded of the view or far away from the camera. Such method would be useful for games, especially Real-Time-Strategy.

Acknowledgment

We gratefully acknowledge the partial financial support from FAPERGS through grant 2283-2551/14-8, and CNPq through grants 305071/2012-2 and 449555/2014-3. Also NVIDIA Corporation for hardware donation.

References

[1] J. van den Berg, S. Patil, J. Sewall, D. Manocha, and M. Lin, "Interactive navigation of multiple agents in crowded environments," in *Proceedings of the 2008 ACM symposium on Interactive 3D graphics and games*, 2008.

[2] S. J. Guy, J. Chhugani, C. Kim, N. Satish, M. Lin, D. Manocha, and P. Dubey, "Clearpath: highly parallel collision avoidance for multi-agent simulation," in *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2009.

[3] J. Snape, J. van den Berg, S. J. Guy, and D. Manocha, "The hybrid reciprocal velocity obstacle," *IEEE Transactions on Robotics*, vol. 27, no. 4, 2011.

[4] M. Kapadia, A. Beacco, F. Garcia, V. Reddy, N. Pelechano, and N. I. Badler, "Multi-domain real-time planning in dynamic environments," in *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2013.

[5] F. M. Garcia, M. Kapadia, and N. I. Badler, "Gpu-based dynamic search on adaptive resolution grids," in *IEEE International Conference on Robotics and Automation*, 2014.

[6] R. P. Torchelsen, L. F. Scheidegger, G. N. Oliveira, R. Bastos, and J. L. Comba, "Real-time multi-agent path planning on arbitrary surfaces," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2010.

[7] H. Jiang, W. Xu, T. Mao, C. Li, S. Xia, and Z. Wang, "Continuum crowd simulation in complex environments," *Computers & Graphics*, vol. 34, no. 5, 2010.

[8] A. Treuille, S. Cooper, and Z. Popović, "Continuum crowds," in *ACM Transactions on Graphics*, vol. 25, no. 3, 2006.

[9] T. Jund, P. Kraemer, and D. Cazier, "A unified structure for crowd simulation," *Computer Animation and Virtual Worlds*, vol. 23, 2012.

[10] B. C. Ricks and P. K. Egbert, "A whole surface approach to crowd simulation on arbitrary topologies," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 2, 2014.

[11] A. Golas, R. Narain, and M. Lin, "Hybrid long-range collision avoidance for crowd simulation," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2013.

[12] J. Ondřej, J. Pettré, A.-H. Olivier, and S. Donikian, "A synthetic-vision based steering approach for crowd simulation," in *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, 2010.

[13] A. Bleiweiss, "Gpu accelerated pathfinding," in *23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2008.

[14] A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *International Parallel and Distributed Processing Symposium*, vol. 28, 2014.

[15] X. Ying, S.-Q. Xin, and Y. He, "Parallel chen-han (pch) algorithm for discrete geodesics," *ACM Transactions on Graphics*, vol. 33, 2014.

[16] B. C. Ricks and P. K. Egbert, "Optimal acceleration thresholds for non-holonomic agents," *The Visual Computer*, no. 6-8, 2014.

[17] S. Singh, M. Kapadia, B. Hewlett, G. Reinman, and P. Faloutsos, "A modular framework for adaptive agent-based steering," in *Symposium on Interactive 3D Graphics and Games*. ACM, 2011.

[18] N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart, "Rectangular multi-chart geometry images," in *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, 2006.

[19] B. Lévy, S. Petitjean, N. Ray, and J. Maillot, "Least squares conformal maps for automatic texture atlas generation," in *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3. ACM, 2002.

[20] K. Hormann, K. Polthier, and A. Sheffer, "Mesh parameterization: theory and practice," in *ACM SIGGRAPH ASIA 2008 courses*.

[21] L. Liu, L. Zhang, Y. Xu, C. Gotsman, and S. J. Gortler, "A local/global approach to mesh parameterization," in *Computer Graphics Forum*, vol. 27, no. 5. Wiley Online Library, 2008.

[22] F. de Goes, K. Crane, M. Desbrun, P. Schröder *et al.*, "Digital geometry processing with discrete exterior calculus," in *ACM SIGGRAPH 2013 Courses*, 2013.

[23] J. van den Berg, S. J. Guy, J. Snape, M. C. Lin, and D. Manocha, "Reciprocal collision avoidance for real-time multi-agent simulation," 2014. [Online]. Available: http://gamma.cs.unc.edu/RVO2/

[24] Project-Team-ALICE, "Opennl (open numerical library)," http://alice.loria.fr/index.php/software/4-library/23-opennl.html, 2014.

[25] P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe, "Texture mapping progressive meshes," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001.