Optimal Path Maps on the GPU

Renato Farias and Marcelo Kallmann

Abstract—We introduce a new method for computing optimal path maps on the GPU using OpenGL shaders. Our method explores GPU rasterization as a way to propagate optimal costs on a polygonal 2D environment, producing optimal path maps which can efficiently be queried at run-time. Our method is implemented entirely with GPU shaders, does not require pre-computation, addresses optimal path maps with multiple points and line segments as sources, and introduces a new optimal path map concept not addressed before: maps with weights at vertices representing possible changes in traversal speed. The produced maps offer new capabilities not explored by previous navigation representations and at the same time address paths with global optimality, a characteristic which has been mostly neglected in animated virtual environments. The proposed path maps partition the input environment into the regions sharing a same parent point along the shortest path to the closest source, taking into account possible speed changes at vertices. The proposed approach is particularly suitable for the animation of multiple agents moving toward the entrances or exits of a virtual environment, a situation which is efficiently represented with the proposed path maps.

Index Terms—Graphics data structures and data types, path planning, shortest path maps, navigation, GPU programming.

1 Introduction

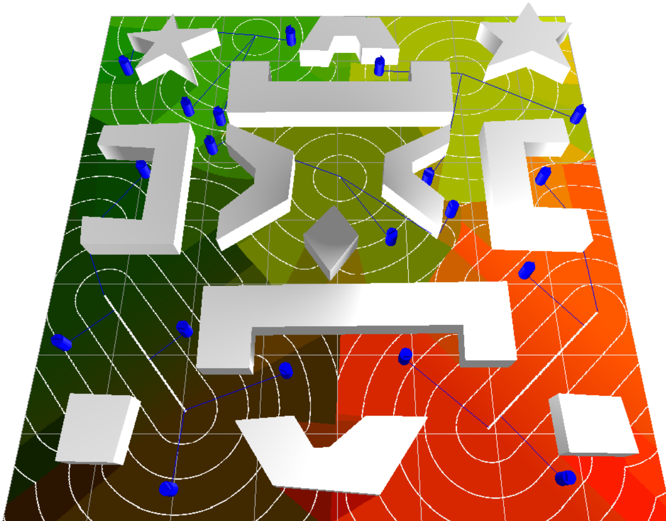


Fig. 1: Example of a multi-source Optimal Path Map computed on a polygonal scene with obstacles. There are three source points in the upper half of the scene, and two line segment sources in the lower half. The contour lines represent equal distances to their closest source. Contour lines are directly extracted from a distance field which is stored in the Z-Buffer as a result of our method. The blue cylinders are agents and each has a polygonal line representing its shortest path to the closest source.

At its core, global navigation often depends on efficient path planning, which is therefore crucial in various applications from planning motions for robots to controlling autonomous agents in virtual environments. This paper focuses on the computation of optimal path maps, which allow for the efficient extraction of optimal trajectories for agents in virtual environments.

While several approaches have been introduced in recent years for computing paths among obstacles, the focus has mostly been on the efficiency of computing collision-free paths without global optimality guarantees. No recent advancement has been achieved on practical methods for computing globally optimal shortest paths, which in regular planar environments are also known as Euclidean shortest paths. This situation reflects the fact that computing optimal paths efficiently is not a trivial task. One popular way of computing Euclidean shortest paths is by constructing a visibility graph of the environment and then running graph search on it [1]. Unfortunately in the worst case the number of edges in the visibility graph is $\Theta(n^2)$, where n is the number of vertices describing the obstacles, which can significantly slow down path queries based on search algorithms running on the graph. Furthermore, each path query requires a new search.

Euclidean shortest paths can however be computed in $\mathcal{O}(n \log n)$ time with Shortest Path Maps (SPMs). SPMs are constructed with respect to a "source point", and like Voronoi diagrams, SPMs partition the space into regions. Whereas regions in Voronoi diagrams share the same closest site, regions in SPMs share the same parent points along the shortest path to the source, which means that an SPM encodes shortest paths between a specified source and *all* other reachable points in a particular planar environment.

While SPMs have been studied in Computational Geometry for several years, they have not been popular in practical applications. This is because their computation involves several complex steps, even when considering non-optimal construction algorithms. The proposed GPU computation approach greatly simplifies the process of building SPMs, allowing them to be easily computed with rasterization procedures triggered from OpenGL shaders without any pre-computation needed. We call our maps Optimal Path Maps (OPMs) because they contain all of the functionality of SPMs and in addition address important extensions: maps with multiple sources of different types and maps representing velocity changes at vertices. See Figures 1 and 6 for examples.

- *R. Farias is with the Computer Science and Engineering Department, University of California, Merced, CA, 95343.*
E-mail: rfarias2@ucmerced.edu
- *M. Kallmann is with the Computer Science and Engineering Department, University of California, Merced, CA, 95343.*
E-mail: mkallmann@ucmerced.edu

Manuscript received in 2018; revised in 2019.

Our approach introduces several advantages. While most representations require a point localization technique in order to determine the region containing the query point, in the proposed approach point localization is reduced to a simple constant time grid buffer mapping. After this mapping, since every point in the OPM has direct access to its parent vertex along the shortest path to the closest source, agents have direct access to the next point to aim when executing their trajectories. In addition, if the entire shortest path is needed, it can be retrieved only in linear time with respect to the number of vertices along the shortest path.

Our approach is based on the idea of cone rasterization from sources and obstacle vertices. Unlike our initial work in this area [2], in the present method we do not require pre-computation of the shortest path tree of the environment and we also do not need to create any geometry for the rasterized cones. Instead we use dedicated fragment shaders to simply fill in the pixels that have direct line-of-sight to the vertices, improving computation speed and also eliminating errors that were introduced from discretizing cone geometry into triangles.

Our shaders operate on the original coordinates of the input vertices for all distance computations; therefore, when the buffer resolution is appropriate (as discussed in Section 6.2), our maps produce exact results not affected by the grid resolution. In addition, our approach allows us to introduce a new type of map not addressed before: maps with weights at vertices, which allow accounting for speed changes at vertices, an interesting situation leading to new types of optimal path maps. Our method can produce relatively complex dynamically-changing OPMs at real-time rates and several examples and benchmarks are presented. Our method can compute paths faster than many competing approaches.

2 Related Work

Our work is related to different areas, from GPU computing and path planning to the computation of distance fields. The related work review below is organized according to these areas.

Approaches to Path Planning Researchers in AI usually approach path planning with discrete search methods on grid-based environments, sometimes making use of hierarchical representations. Several advancements on discrete search methods have been proposed such as heuristic search, dynamic replanning, anytime planning [3], etc.; however, with few attempts to approximate Euclidean shortest paths. One exception is the work on "any-angle path planning" [4], which significantly improves paths computed on grids with respect to getting close to a global optimal; however, not guaranteeing to achieve globally-shortest paths.

In Computer Animation, while several approaches have been introduced in recent years for efficiently computing paths among obstacles, the state-of-the-art has focused mostly on the efficiency of computing collision-free paths. For instance, recent work has addressed new definitions of navigation meshes [5] [6] [7] but mostly addressing contributions related to speed of computation and computing paths with clearance. Given the complexity of the problem and the high computational cost of the simple approaches to it, global optimality is simply not addressed.

One way to compute globally-optimal Euclidean shortest paths is to first build the visibility graph [8] [9] [10] of the environment and then run a graph search algorithm on it [11] [1]. Previous work [12] has presented specific cases where the problem can be solved with $\mathcal{O}(n \log n)$ time algorithms without explicitly building the entire visibility graph. However, a visibility graph can have $\Theta(n^2)$ edges, where n is the number of vertices describing the environment, making it expensive to be computed, updated and queried. In addition, a new graph search is needed for each path query. It is therefore difficult to develop efficient methods based on visibility graphs.

Shortest Path Maps The first proposed method based on Shortest Path Maps (SPMs) has worst-case time complexity of $\mathcal{O}(kn \log^2 n)$ [13], where k is a parameter called the "illumination depth", which is bounded above by the number of different obstacles touching a shortest path. Later, the first worst-case sub-quadratic algorithm for Euclidean shortest paths was proposed applying the continuous Dijkstra expansion, which naturally leads to the construction of SPMs [14]. The continuous Dijkstra technique simulates expanding wavefronts, which are the set of all points equally distant from a given source point. The expansion requires solving various events such as wavefront self-collisions forming hyperbolic boundaries. The result of the wavefront propagation is a spatial partition which is the SPM.

An algorithm for computing SPMs has been proposed taking $\mathcal{O}(n \log n)$ time to preprocess the environment, allowing distance-to-source queries to be answered in $\mathcal{O}(\log n)$ time, and paths to be returned in $\mathcal{O}(\log n + k)$ time, where k is the number of turns along the path [15]. Unfortunately, these methods and all the known algorithms with good theoretical running times involve complex techniques and data structures that overburden their practical implementation in applications and prevent the development of useful extensions. In contrast, our GPU-based approach is relatively simple and is less affected by typical robustness difficulties encountered in many geometric computations for building spatial subdivisions relying only on floating point operations.

Alternative GPU approaches have also been explored in previous work. The first attempt to compute SPMs in GPU was designed to take advantage of the GPU's massive parallelization capabilities [16]. The method first pre-computes in CPU the visibility graph and the shortest path tree (SPT) of the environment. Afterwards, a brute-force but parallelized GPU computation is used to determine the SPT vertex to connect to each pixel in order to produce a subdivision of the discrete screen space in SPM regions.

The approach of using shader rasterization as an efficient way to propagate wavefronts in the GPU was introduced in our previous work [2] and in this work we present a completely re-designed method incorporating several extensions and significantly improving the approach in multiple ways: 1) we eliminate the need to precompute the visibility graph and SPT, 2) in doing so we are able to address segment sources and speed changes at vertices, and 3) we no longer have to construct actual geometry for the rendered cones simulating wavefront expansions; instead we simply employ a dedicated fragment shader to directly fill in the relevant pixels, simplifying the process and most importantly eliminating error accumulation from cone discretization.

Other GPU Methods Previous work has investigated computing geodesic distances in parallel on GPUs [17] [18] as well as rasterization-based GPU techniques for related applications such

as for computing Voronoi diagrams [19]. Although we also employ rasterization techniques to accumulate distances, our approach introduces the significant insight of placing clipped primitives at accumulated heights in order to compute a SPM. Furthermore, we employ new techniques taking advantage of modern programmable shaders without discretizing geometry during the process, thus eliminating errors from geometry discretization.

GPU methods have also been explored for path planning from grid-based searches, for example by performing multiple short-range searches in parallel [20], by parallelizing expansions per-pixel on uniform grids [21] and by relying on a quad-tree scheme [22]. However, grid-based approaches do not address optimality in the Euclidean sense. We nevertheless compare reported times from some of these works with our approach (Table 2) and show that in addition to global optimality our method is also faster in many cases.

Beyond path planning, GPU methods reported in the literature have also addressed Delaunay triangulations [23], local collision avoidance [24], shortest path trees on graphs [25], and breadth-first search (BFS) on graphs [26] [27]. In particular, the BFS method of Merrill et al. [26] is highly efficient and, although not directly applicable to computing Euclidean shortest paths or SPMs, we also include it in our comparative table (Table 2).

Distance Fields on Meshes Computing distance fields is a problem closely related to computing SPMs. While these methods do not represent the boundaries of a SPM decomposition, many of the methods could be extended to do so. Previous work including methods for computing geodesic distances and distance fields are numerous and we include here an overview of the area.

A popular method to compute distance fields is to rely on window propagation on meshes. The approach of Mitchell et al. [28] propagates front windows in unfolded triangles while solving front events during propagation, taking $\mathcal{O}(n^2 \log n)$ time to compute geodesics. Implementations of the approach are available [29] and extensions addressing arbitrary polygons on a mesh have also been explored [30]. It is also possible to perform window propagation without handling all events [31] reaching $\mathcal{O}(n^2)$ time but in practice processing a high amount of overlapping windows. Window pruning techniques have then been investigated to improve practical running times [32] [33], making window propagation an efficient approach for computing exact geodesics. We have compared our GPU approach against the window propagation method of Xin and Wang [32] as implemented in CGAL. Our results show that while our approach was slower to compute our SPM representation versus their sequence tree, it was faster in querying paths (see Table 3). Efficient front propagation methods based on the Fast Marching Method have also been proposed [34] [35] [36]; however, unlike the window propagation methods discussed above, they are limited to producing approximate geodesics.

Among other related methods, previous work has already addressed multiple types of sources, for instance when computing geodesic Voronoi diagrams with multiple sources [37] as well as polyline-sourced Voronoi diagrams [38] [39]. Such concepts however have not been applied to shortest path maps.

Summary While several algorithms exist for computing shortest path maps, available methods are either too complex for practical use or too expensive for real-time applications. The proposed method is the first to be implemented entirely with GPU shaders, it does not require any pre-computation, it addresses new capabilities not explored by previous navigation representations, and it enables multi-agent navigation based on paths with global optimality, a characteristic which has been neglected in simulated virtual environments developed to date. While powerful related methods in the geometry processing area are available for computing distance fields, they have not been applied to represent SPM boundaries or to compute paths with speed changes. To our knowledge our work has produced SPM diagrams of complexity not seen before in previous work. See for example Figure 17.

3 Multi-Source Optimal Path Maps

We first describe the base OPM case with multiple source points. Let n_s be source points $\{\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_{n_s}\}$ in the plane, such that $\mathbf{s}_i \in \mathcal{D}$, $i \in \{1, 2, ..., n_s\}$, and where $\mathcal{D} \subset \mathbb{R}^2$ defines a polygonal domain containing all sources. In all our examples $\mathcal{D}$ is a rectangular area delimiting the environment of interest, and the GPU framebuffer will be configured to entirely cover $\mathcal{D}$. A set of polygonal obstacles $\mathcal{O}$, with a total of n vertices, is also defined in $\mathcal{D}$ such that shortest paths will not cross any obstacles in $\mathcal{O}$.

Given source points the respective OPM will efficiently represent globally-shortest paths $\pi^*(\mathbf{p})$, which are optimal collision-free paths from any point $\mathbf{p} \in \mathcal{D} - \mathcal{O}$ to its closest source point $\mathbf{s}_i$, in the sense that $\mathbf{s}_i = min_j \lambda^*(\mathbf{p}, \mathbf{s}_j)$, where $\lambda^*(\mathbf{p}, \mathbf{s}_j)$ denotes the length of the shortest path $\pi^*(\mathbf{p}, \mathbf{s}_j)$, $j \in \{1, 2, ..., n_s\}$. Our OPM also efficiently represents the values of λ^* for all pixels of the framebuffer by storing them in a dedicated buffer created in the OpenGL pipeline. This representation gives us direct access to the distance field of the environment and allows us to easily draw the white isolines that can be seen in most of the figures in this paper. Depending on the situation source points can represent the start or the end point of a path. In most of the presented examples sources will represent goals to be reached by agents placed anywhere in the environment.

The plane represented by the framebuffer is located at $z = 0$. The basic idea of our method is to rasterize "clipped cones" with apices placed below source points and obstacle vertices, at the correct z heights, so that the final rendered result from an orthographic top-down view is the desired OPM (see Figure 2).

The process is implemented as follows. An array containing the n_s source points and n obstacle vertices is stored in the GPU. At each iteration one point or vertex is copied into a reserved position of a data array where it will be used to rasterize a clipped cone. The point or vertex that is selected to generate the clipped cone at each iteration is referred to as that iteration's "generator." Each point and vertex is processed once, such that the result is given after $n_s + n$ iterations.

Important to our approach is the fact that we do not actually need to create discretized geometry for representing and then drawing cones. Instead, we simply fill in pixels that have direct line-of-sight to the generator, which is an equivalent operation. A cone apex is located below the generator relative to the $z = 0$ plane. The depth values of the affected pixels increase proportionally to their Euclidean distances to the apex, as with the slope of a cone. Because the depth is accumulated over iterations, it represents the distance back to the source point along the shortest path, λ^*.

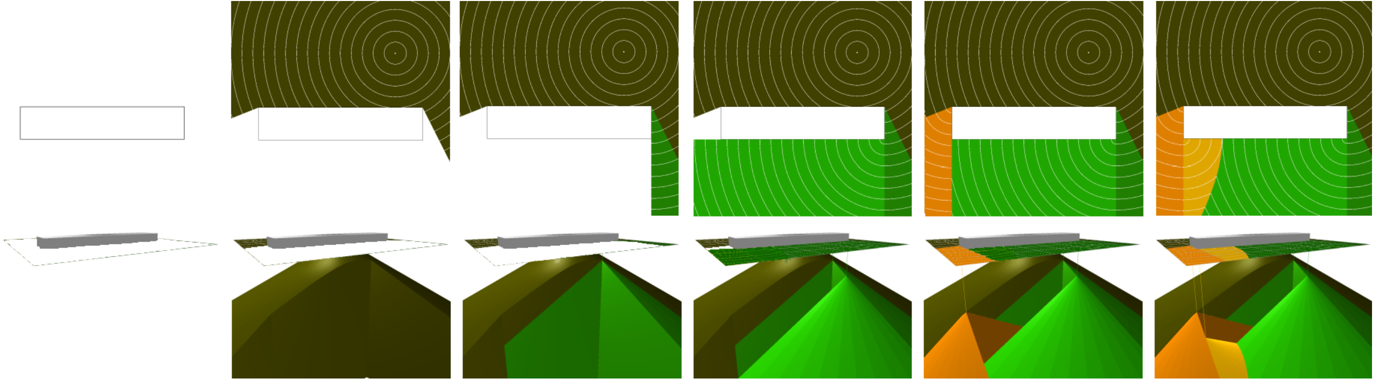


Fig. 2: Top row: example steps for computing a single-source SPM. Bottom row: corresponding 3D perspective view of each step.

When all clipped cones are drawn at their respective heights, the GPU's depth test will maintain, for each pixel, the correct parent generator point, which is the immediate next point on the shortest path from that pixel to the closest source point. We say that a cone "loses" to another at a given pixel when its depth at the pixel is greater than the depth generated from another cone. The GPU depth's test automatically discards pixel associations to cones that lose to other cones.

3.1 Method Description

Given polygonal obstacles $\mathcal{O}$ with n vertices and n_s source points, $n_s \geq 1$, the total number of vertices to be processed is $n_{total} = n + n_s$. These vertices are stored in array DATAARRAY of size $n_{total} + 1$. The extra position is reserved for storing at each iteration the current generator that will be used for cone rasterization. By convention this is the first position in the array, DATAARRAY[0], and will be referred to as $\mathbf{g}_{cur}$. Once DATAARRAY is constructed, it is stored in the GPU as a Shader Storage Buffer Object. Each of the $n_{total} + 1$ positions in DATAARRAY stores:

- x, y: The original coordinates of the point or vertex in $\mathcal{D}$.
- STATUS: A flag that can be equal to SOURCE for sources, OBSTACLE for obstacle vertices, or EXPANDED for points or vertices which have already generated a cone.
- DISTANCE: The current known shortest path distance to the closest source point, λ^*. This will always be 0 for source points and is initially undetermined for obstacle vertices.
- PARENTID: Array index into DATAARRAY of the current parent point, which is the next point on the shortest path back to the closest source point. Since sources have no parent point, by convention they simply store their own index.

The framebuffer stores similar information for the pixels. For each pixel, its red and green components store the x and y coordinates of its parent point (equivalent to DATAARRAY[PARENTID]$.xy$), its blue component stores λ^* (equivalent to DISTANCE), and its alpha channel stores either 0 if the pixel has yet to be reached by a cone or >0 otherwise. When the buffer is drawn, the color of each pixel is mapped in the following way: x is used as the red component, y is used as the green component, and the blue component is zeroed. Although this mapping is arbitrary, it allows to visualize the location of a region's parent from the red and green intensities.

The OPM generation consists of four steps which repeat n_{total} times such that each point and vertex is processed once. The steps are presented in Procedures 1-4. The hat notation (e.g., $\widehat{\mathbf{n}}$) denotes unit vectors.

Step 1 is a search in DATAARRAY where the position with the smallest DISTANCE is copied into the reserved position of the array, index 0. Only points or vertices which have not yet generated a cone (STATUS≠EXPANDED) are considered in this search, and once one is chosen its status is updated to EXPANDED so that it cannot be processed again. The point that is chosen becomes $\mathbf{g}_{cur}$, the current generator. This step can be skipped in the first iteration of the algorithm as we can just start with one of the source points.

Procedure 1 Search Compute Shader

Input: DATAARRAY
1: int $generatorId \leftarrow -1$;
2: float $generatorDist \leftarrow \infty$;
3: **for** ($\forall i, i \in 1, 2, ..., n_{total}$) **do**
4: **if** (DATAARRAY[i].STATUS $\neq$ EXPANDED) **then**
5: **if** (DATAARRAY[i].STATUS = SOURCE **or** $generatorId = -1$ **or** DATAARRAY[i].DISTANCE $<$ $generatorDist$) **then**
6: $generatorId \leftarrow i$;
7: $generatorDist \leftarrow$ DATAARRAY[i].DISTANCE;
8: **end if**
9: **end if**
10: **end for**
11: DATAARRAY[0] $\leftarrow$ DATAARRAY[$generatorId$];
12: DATAARRAY[$generatorId$].STATUS $\leftarrow$ EXPANDED;

Step 2 is to generate a shadow area in order to solve visibility constraints. Using a geometry shader, we draw into a stencil buffer three triangles behind every obstacle line segment that is front-facing with respect to $\mathbf{g}_{cur}$, in a manner illustrated in Figure 3. Any pixel covered by one of these triangles is considered to be in shadow. The resulting buffer is used as a stencil buffer in the next step. Three triangles is the minimum number of triangles needed to cover all possible shadow shapes. We use constant $c_{svf} > 0$, which stands for shadow vector factor, when computing the points that make up the triangles. This constant must be large enough to handle shadows of all sizes. Since our coordinates are OpenGL normalized coordinates in the $[-1, 1]$ range, a value of 4 is always enough. Note that limiting shadows to front-facing

segments is merely for efficiency; generating triangles behind back-facing segments would not affect the shadow area.

Step 3 draws a clipped cone with the generator $\mathbf{g}_{cur}$ directly above its apex along the z axis. As previously stated, we do not actually create geometry for the cone but instead simply run a fragment shader over every pixel on the screen. The pixels that are not in shadow have direct line-of-sight to $\mathbf{g}_{cur}$, so they calculate their Euclidean distance to $\mathbf{g}_{cur}$ and add it to $\mathbf{g}_{cur}$'s accumulated distance, DISTANCE. If this sum is smaller than the current DISTANCE of the pixel (from the cone of a previous $\mathbf{g}_{cur}$), then its DISTANCE is updated and its PARENTID is set to $\mathbf{g}_{cur}$'s index.

Finally, step 4 is to update the DISTANCE of all vertices visible from the current generator, in a way similar to step 3. Each vertex not in shadow calculates its distance to $\mathbf{g}_{cur}$ plus $\mathbf{g}_{cur}$'s DISTANCE, and if that sum is smaller than its previous DISTANCE it stores the new DISTANCE and $\mathbf{g}_{cur}$'s index in its PARENTID. The reason steps 3 and 4 are separate is because step 3 is updating the framebuffer, while step 4 is updating DATAARRAY. The end of this step is a synchronization point in our GPU implementation.

After all points and vertices have been processed, which means n_{total} iterations of steps 1-4, the result in the framebuffer will be the desired OPM. Examples of OPMs with a single source point are shown in Figure 15 and with multiple source points are shown in Figure 16.

Procedure 2 Shadow Area Geometry Shader

Input: DATAARRAY
Input: $\mathbf{g_{cur}}$ *// Current generator point/vertex*
Input: e *// One of the sides of a scene obstacle*
1: vec4 $\mathbf{p}_1 \leftarrow$ first endpoint of e;
2: vec4 $\mathbf{p}_2 \leftarrow$ second endpoint of e;
3: vec4 $\mathbf{p}_m \leftarrow (\mathbf{p}_1+\mathbf{p}_2)/2$;
4: vec4 $\mathbf{p}_g \leftarrow$ project and normalize vec4($\mathbf{g_{cur}}.xy$, 0, 0);
5: float $dx \leftarrow \mathbf{p}_2.x - \mathbf{p}_1.x$;
6: float $dy \leftarrow \mathbf{p}_2.y - \mathbf{p}_1.y$;
7: vec4 $\widehat{\mathbf{g}} \leftarrow$ normalize($\mathbf{p}_m - \mathbf{p}_g$);
8: vec4 $\widehat{\mathbf{n}} \leftarrow$ normalize(vec4($dy, -dx, 0, 0$));
9: float $d \leftarrow$ dot($\widehat{\mathbf{g}}$, $\widehat{\mathbf{n}}$);
10: **if** ($d < 0.1$) **then**
11: vec4 $\widehat{\mathbf{v}}_1 \leftarrow$ normalize($\mathbf{p}_1 - \mathbf{p}_g$);
12: vec4 $\widehat{\mathbf{v}}_2 \leftarrow$ normalize($\mathbf{p}_2 - \mathbf{p}_g$);
13: vec4 $\mathbf{p}_{1s} \leftarrow \mathbf{p}_1 + c_{svf}\widehat{\mathbf{v}}_1$;
14: vec4 $\mathbf{p}_{2s} \leftarrow \mathbf{p}_2 + c_{svf}\widehat{\mathbf{v}}_2$;
15: vec4 $\mathbf{p}_{ms} \leftarrow \mathbf{p}_m + c_{svf}\widehat{\mathbf{g}}$;
16: EmitPrimitive($\mathbf{p}_1$, $\mathbf{p}_{ms}$, $\mathbf{p}_{1s}$);
17: EmitPrimitive($\mathbf{p}_2$, $\mathbf{p}_{2s}$, $\mathbf{p}_{ms}$);
18: EmitPrimitive($\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_{ms}$);
19: **end if**

3.2 Time Complexity

Given that the number of sources n_s can be considered to be constant, steps 1-4 are repeated $\Theta(n)$ times in our overall method. Step 1 is a loop where the next generator among the input vertices is searched, taking time $\mathcal{O}(n)$. Step 2 renders the shadow area for each generator, and in the worst case an area covering most of the framebuffer is generated, taking time $\mathcal{O}(nr)$, where r is the resolution (number of pixels) in the framebuffer. Step 3 renders a clipped cone in a single rendering pass over the framebuffer, taking time $\mathcal{O}(r)$. Step 4 updates the distances of all visible obstacle vertices, taking time $\mathcal{O}(n)$. The overall SPM construction time is therefore $\mathcal{O}(rn^2)$. A discussion on the requirements for the framebuffer resolution is given in Section 6.2. Note however that steps 2, 3, and 4 are executed in parallel on the GPU, and thus the final running time depends on the number of GPU cores available. Step 4 in particular can be reduced to $\mathcal{O}(1)$ if the number of cores is enough to process in parallel the maximum number of visible vertices from a generator. Considering c to be the number of GPU cores available the overall construction time will be $\mathcal{O}(\frac{r}{c}n^2)$.

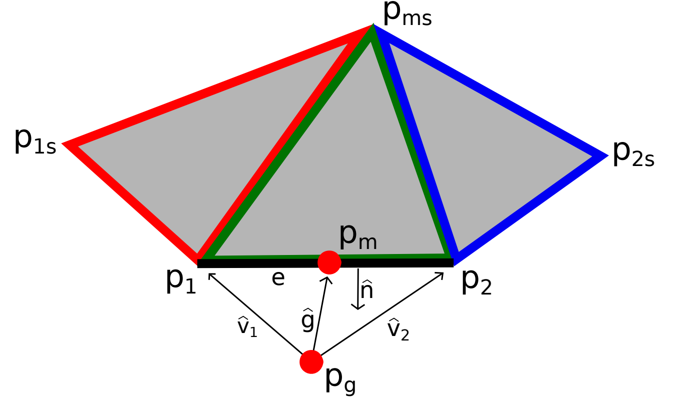


Fig. 3: Example of a shadow area. The line segment e represents the side of an obstacle. The red point $\mathbf{p}_g$ is the generator, points $\mathbf{p}_1$ and $\mathbf{p}_2$ are the endpoints of e, and point $\mathbf{p}_m$ is the middle point of e. Vectors $\widehat{\mathbf{v}}_1$, $\widehat{\mathbf{v}}_2$, and $\widehat{\mathbf{g}}$ are the normalized vectors from $\mathbf{p}_g$ to $\mathbf{p}_1$, $\mathbf{p}_g$ to $\mathbf{p}_2$, and $\mathbf{p}_g$ to $\mathbf{p}_m$, respectively. Points $\mathbf{p}_{1s}$, $\mathbf{p}_{2s}$, and $\mathbf{p}_{ms}$ are calculated in the following way: $\mathbf{p}_{1s} = \mathbf{p}_1 + c_{svf}\widehat{\mathbf{v}}_1$, $\mathbf{p}_{2s} = \mathbf{p}_2 + c_{svf}\widehat{\mathbf{v}}_2$, and $\mathbf{p}_{ms} = \mathbf{p}_m + c_{svf}\widehat{\mathbf{g}}$. The three triangles are sufficient to cover the entire area behind the segment. Using less than three triangles may not result in a correct shadow if the generator is close to the segment because the area becomes wide and thin. Value 4 is used for constant c_{svf} such that shadows of any size can be handled given that our obstacle coordinates are normalized.

Procedure 3 Cone Fragment Shader

Input: DATAARRAY
Input: $\mathbf{g_{cur}}$ *// Current generator point/vertex*
Input: $fragCoord$ *// xy coordinates of the pixel*
Output: vec4 $fragValue$
1: bool $inShadow \leftarrow$ is the pixel in shadow or not?;
2: vec4 $currentValue \leftarrow$ what's currently stored in this pixel; *//texture fetch*
3: vec4 $fragValue \leftarrow currentValue$; *// If nothing else, pass the current value on*
4: **if** ($inShadow$ = false) **then**
5: vec2 $\mathbf{p} \leftarrow$ normalize $fragCoord$;
6: vec2 $\mathbf{p}_g \leftarrow$ project and normalize $\mathbf{g_{cur}}.xy$;
7: float $newDist \leftarrow$ distance($\mathbf{p}$, $\mathbf{p}_g$) + $\mathbf{g_{cur}}$.DISTANCE;
8: **if** (there is no currently stored distance in the pixel **or** $newDist < currentValue.z$) **then**
9: $fragValue \leftarrow$ vec4($\mathbf{g_{cur}}.xy, newDist, 1$);
10: **end if**
11: **end if**

Procedure 4 Distance Compute Shader

Input: DATAARRAY
Input: $\mathbf{g_{cur}}$ // *Current generator point/vertex*
1: int $id \leftarrow$ index of the vertex to be updated;
2: bool $inShadow \leftarrow$ is the vertex in shadow or not?;
3: **if** ($inShadow$ = false) **then**
4: vec2 $\mathbf{p} \leftarrow$ project and normalize DATAARRAY$[id].xy$;
5: vec2 $\mathbf{p}_g \leftarrow$ project and normalize $\mathbf{g_{cur}}.xy$;
6: float $newDist \leftarrow$ distance($\mathbf{p}$, $\mathbf{p}_g$) + $\mathbf{g_{cur}}$.DISTANCE;
7: **if** (there is no currently stored distance in DATAARRAY$[id]$ **or** $newDist <$ DATAARRAY$[id]$.DISTANCE) **then**
8: DATAARRAY$[id]$.DISTANCE $\leftarrow newDist$;
9: DATAARRAY$[id]$.PARENTID $\leftarrow \mathbf{g_{cur}}$'s original index;
10: **end if**
11: **end if**

Once the OPM is constructed queries can be performed efficiently. Query points are located in the framebuffer in $\mathcal{O}(1)$ time, path length queries to the closest source are answered in $\mathcal{O}(1)$ time, and paths to the closest source are reconstructed in time $\Theta(k)$, where k is the number of vertices in the returned polygonal path.

4 SEGMENT SOURCES

Line segment sources are one natural extension to our method, and are interesting as sources for what they can represent. Many navigation goals in real-world scenarios are not single points but segments, such as the finish line of a race, the thresholds of doorways or hallways, or the boundary of a coastline. Many of these cases appear when planning evacuation routes from buildings. Being able to compute OPMs with segments as sources allows us to maintain global optimality in these practical situations.

Consider that we now have n_l line segment sources $\{\mathbf{l}_1, \mathbf{l}_2, ..., \mathbf{l}_{n_l}\}$, such that $\mathbf{l}_i$, $i \in \{1, 2, ..., n_l\}$, consists of two endpoints $\in \mathcal{D}$. The OPM will then efficiently represent globally-shortest paths $\pi^*(\mathbf{p})$, which are now optimal collision-free paths from any point $\mathbf{p} \in \mathcal{D} - \mathcal{O}$ to the closest reachable point on a segment source $\mathbf{l}_i$.

Every line segment $\mathbf{l}_i$ can have n_{c_i} *critical points*, $n_{c_i} \geq 0$. A critical point denotes a point on the segment onto which at least one obstacle vertex projects. The obstacle vertex must have direct line-of-sight to the segment. Critical points are where the visibility of the scene changes with respect to the segment and are useful because in practice every path that passes through the corresponding obstacle vertex will have its shortest path reach the line segment on that critical point. See Figure 4. For each $\mathbf{l}_i$, first the two endpoints of the segment create two entries in DATAARRAY which are treated identically to source points. Then, $n_{c_i} + 1$ further entries are created, where n_{c_i} is equal to the number of critical points segment $\mathbf{l}_i$ possesses. Every one of these entries stores two pairs of xy coordinates rather than just one, with STATUS set to SOURCESEGMENT, to represent the sub-segments of $\mathbf{l}_i$. If $n_{c_i} = 0$, then the two endpoints are simply used because the segment has no sub-segments. If $n_{c_i} > 0$, then every adjacent pair of points, including both endpoints and critical points, will create an entry in DATAARRAY. Coincident critical points are not added and have no effect on the method.

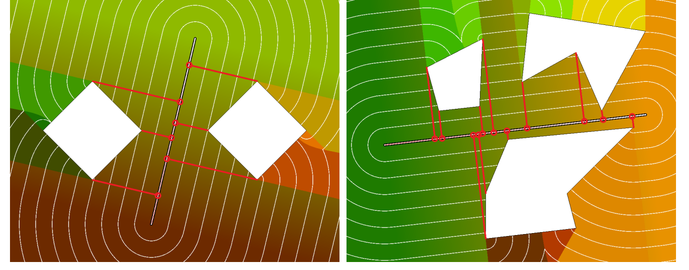


Fig. 4: The circled points on the segment sources are the critical points, which are projections of obstacle vertices.

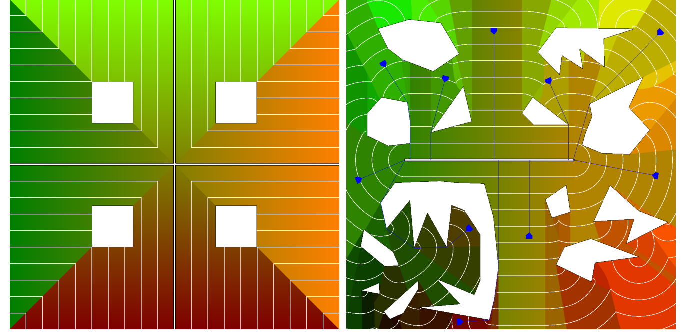


Fig. 5: Line segment source examples. Left: SPM of two segment sources intersecting at the center. Right: Several paths from agents represented as blue triangles to their closest points in a segment source. In both cases the white contours represent the distance field from the sources.

The distance calculation of the OPM generation process is different when the generator's STATUS is marked as SOURCESEGMENT. It is necessary to determine whether the point being updated is closer to one of the endpoints of the sub-segment, or somewhere inbetween. If it is closer to one of the endpoints, the distance is simply the distance to that endpoint. Otherwise, the distance is equal to the distance between the point and its projection on the sub-segment.

The described changes are sufficient to handle both segments and points as sources. Figure 5 shows additional examples of OPMs with line segment sources.

5 VERTEX WEIGHTS

Another useful extension is to consider weights assigned to the vertices of the scene. A weight w on a vertex signifies that when an agent passes by the vertex its speed is changed according to w, implying that the distance calculation for that particular generator's cone will be altered by a certain multiplicative factor which is given by the value of w. This is the equivalent of changing the slope of the cone being rasterized, which is also equivalent to changing the speed at which that wavefront propagates.

As with segment sources, vertex weights allow our maps to represent practical scenarios that have not been explored previously. As an example, consider a virtual character that needs to arrive at a certain destination. One option is to walk directly there; another is to take a more roundabout path that at a certain point lets the character get on a bicycle or another vehicle, speeding up the

traversal of the remaining distance. This scenario is illustrated in Figure 6. A shortest path map cannot answer which option is faster because it cannot represent the change in speed, but an optimal path map considering vertex weights can.

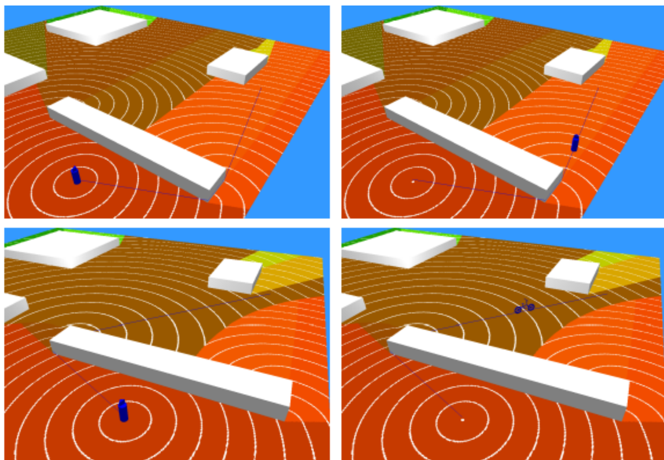


Fig. 6: Top: an agent on foot plots a constant-speed shortest path. Bottom: the top-left vertex of the long rectangular obstacle has its weight increased representing the possibility of using a bicycle to speed-up traversal time. That possibility leads to the fastest path.

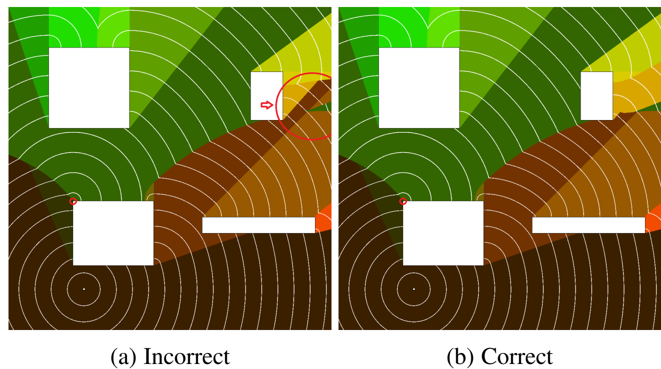
Given two points, $\mathbf{p}_i$ and $\mathbf{p}_j \in \mathcal{D} - \mathcal{O}$, the Euclidean distance between them, $d(\mathbf{p}_i,\mathbf{p}_j)$, and a weight w, $w > 0$, let the weighted distance be equal to $d_w(\mathbf{p}_i, \mathbf{p}_j) = d(\mathbf{p}_i, \mathbf{p}_j)/w$. When all vertices have $w = 1$, a regular OPM is generated. When any vertex has $w \neq 1$, the OPM is altered. For example, if a vertex has $w = 2$, agents that pass through the vertex would move twice as fast. It represents the agent switching to a faster mode of travel.

If a generator with $w \neq 1$ becomes the parent of another vertex with $w' \neq 1$, then the weight stored in the vertex will be $max(w, w')$. This symbolizes the agent always preferring to stick with the fastest mode of travel that it comes across.

For every unique weight that exists in the scene, we must store an extra copy of each of the obstacle vertices in the data array. In a regular OPM it is impossible for a cone to lose to another in close distance but win over a long distance, so there is no need to propagate any but the closest cone for each vertex. In a weighted OPM this is however possible. A cone with a wider slope (higher w) may eventually poke out from under a cone with a narrower slope (lower w). It is thus necessary to propagate the closest cone for every unique weight, otherwise the resulting map may generate incorrect discontinuities.

In the example of Figure 7 the highlighted circled vertex (on the lower-left image quadrant) has a weight of 1.3, making it a more attractive option for optimal paths and thus distorting the OPM towards it. However, as can be seen in (a), it generates a discontinuity on the other side of the map (region highlighted with an arrow) because it was unable to propagate to the area behind the upper-right obstacle. By giving an extra space for the uniquely weighted vertex to propagate (b), an extra cone is drawn and the wavefronts line up correctly.

Interestingly the isolated green region, the one which is not adjacent to a generator vertex as indicated with the arrow in Figure 7, still appears in the correct version of the map. This indicates that if the agent is located in that region, the shortest path to the source first goes to the parent point of the green region, which is disconnected from the isolated region. Therefore OPMs with speed changes do not have anymore the property that each region associated with a parent generator is singly-connected.



(a) Incorrect (b) Correct

Fig. 7: If generators with different weights are not allowed to propagate, they may generate inconsistencies in the OPM. It is necessary to store an extra copy of the data array for each uniquely weighted vertex in the scene, otherwise situations such as the above arise.

The inclusion of vertex weights requires additional solutions for the correct visualization of the obtained OPMs. Consider the example shown in Figure 8. Although the selected goal points (red crosses) in both (a) and (b) have the same parent, the paths that they generate are not the same; one passes through the weighted vertex and the other does not. Here we use dashed lines to differentiate regions with paths altered by the weighted vertex. The inclusion of additional weighted vertices would require additional patterns in the visualization. Because weighted vertices change the speed at which an agent traverses, white isolines no longer denote equal distance to the source but rather equal time intervals. Figure 9 shows the resulting OPM when the weight of the same vertex as in Figure 7 is set to increasing values.

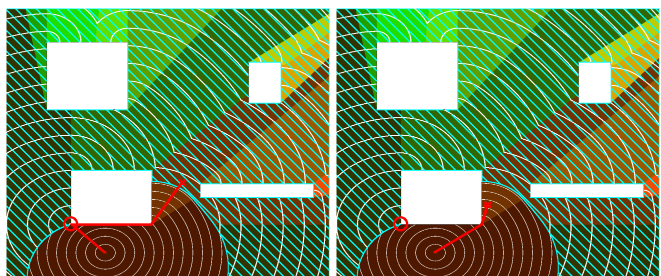
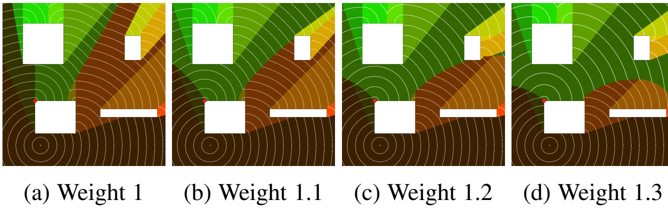


Fig. 8: The weight of the highlighted vertex is increased. Paths to regions with cyan dashed lines pass through the "faster vertex". With vertex weights, ambiguity is introduced because while pixels in the same region still have the same parent, their paths after the parent may be different. This happens because if the goal is sufficiently distant it becomes advantageous to take a detour to pass through the weighted vertex and gain faster travel mode.

6 Results and Discussion

We have produced several agent simulations taking advantage of the new capabilities introduced in this paper. The presented results



(a) Weight 1 (b) Weight 1.1 (c) Weight 1.2 (d) Weight 1.3

Fig. 9: Resulting OPMs as the highlighted vertex has its weight increased. The region generated by the clipped cone at that vertex gradually bloats outwards until it reaches both sides of the map.

are further demonstrated in the accompanying video to this paper.

Dynamically Changing Sources Figure 10 depicts the layout of a subway, with the sources symbolizing train doors which dynamically change from non-existent to points and then growing line segments as the doors open. The OPM is updated in real-time as this happens and the paths of the agents adjust accordingly.

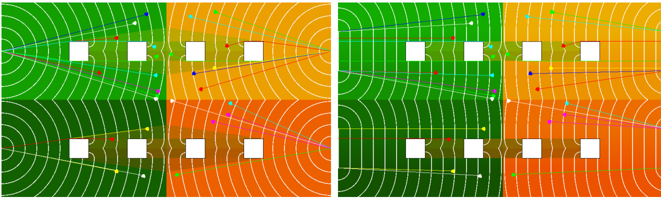


Fig. 10: Simulation dynamically updating an OPM as its sources change from points (left) to growing line segments (right).

Moving Segment Sources We add motion to the segment sources in the simulation depicted in Figure 11. The segment sources represent dynamic goals (trains) that agents attempt to reach. The trains move either left or right on their tracks while the map is continuously updated.

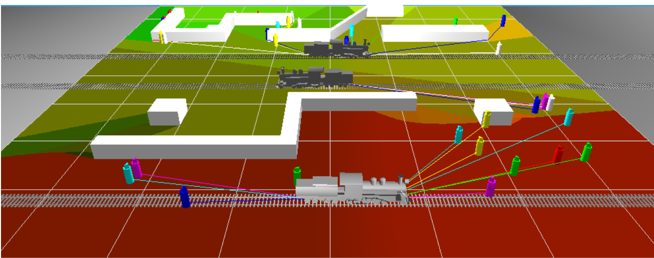


Fig. 11: Simulation where agents attempt to reach moving trains represented by dynamic segment sources. As the trains move each agent has direct access to a shortest path to the closest train.

Evacuation Analysis An OPM is used in Figure 12 to calculate a distance field where a greener color indicates closer proximity to a source while a redder color indicates greater distance from a source. Sources are segments indicating road exits and the illustrated map is a region of the roads in the northwest area of Bodie, CA. Three segment sources represent the exits, one in the northwest area and two in the northeast area. By varying the number of passages leading out of the central area it is possible to visually analyze differences in evacuation distances and the OPM boundaries delimiting different directions towards closest exits. The encoded optimal paths are readily available for simulating autonomous agents.

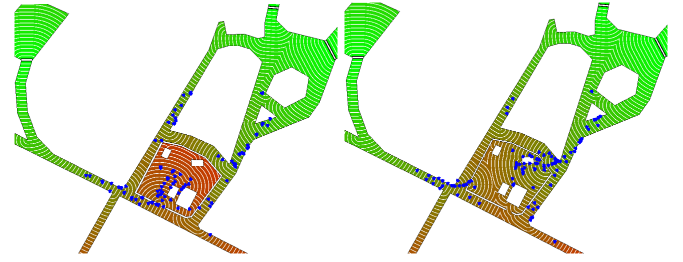


Fig. 12: Evacuation simulation. Left: central area has 1 exit. Right: 3 exits.

Multiple Vertex Weights Multiple vertices of the environment in Figure 13 had their weights increased due the availability of faster transportation modes. Four agents with identical start and end points navigate the environment, one at a time. Each time an agent passes through a weighted vertex it uses the transportation resource and the corresponding weight is reverted to regular, altering the OPM and the fastest paths for subsequent agents.

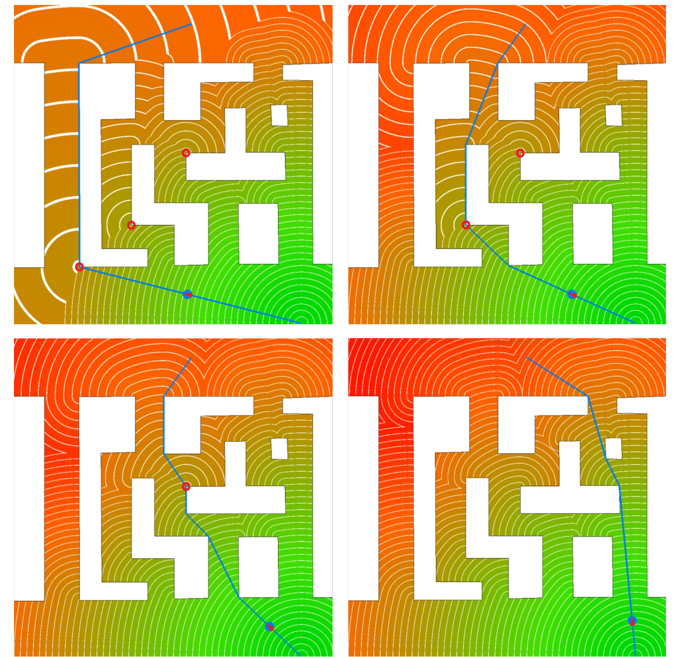


Fig. 13: Simulation with multiple vertex weights.

Map name	P	V	Computation Time (s)	Comp.+Transfer Time (s)
Concave1	2	12	0.0011	0.0207
Concave2	13	96	0.0088	0.0465
Spiral	1	38	0.0022	0.0274
SpmEx1	3	15	0.0016	0.0215
SpmEx2	13	91	0.0100	0.0470
Profiling0	4	16	0.0014	0.0221
Profiling1	16	64	0.0054	0.0404
Profiling2	36	144	0.0456	0.0858
Profiling3	64	256	0.1251	0.1680
Profiling4	100	400	0.2701	0.3099
Profiling5	196	784	0.8564	0.8863
Profiling6	400	1600	2.7371	2.8070

TABLE 1: Average time in seconds to compute a single-source OPM on various maps (shown in Figure 15). P and V are the number of polygons and vertices.

Method	CPU	GPU	Resolution	O	V	Optimality	Time (s)
Dynamic Search using uniform grid (1)	–	GF GT 650M	1024x1024	–	–	Average	32.93
Dynamic Search using uniform grid (1)	–	GF GTX 680	1024x1024	–	–	Average	21.25
Dynamic Search using uniform grid (2)	–	–	1024x1024	–	–	Average	14.12
Dynamic Search using quad-tree (2)	–	–	1024x1024	–	–	No	0.04
Breadth-first search (BFS) on grid-based graph (3)	–	GF GTX 280	1M	–	–	Average	0.17
Shortest Path Map (4)	i7 2.66 GHz	GF GTX 580	1024x1024	64	256	Best	1.42
Optimal Path Map (5)	i7 3.40 GHz	GF GTX 970	1000x1000	64	256	Best	0.13

TABLE 2: A comparison of GPU-based techniques: (1) dynamic search using an uniform grid [21], (2) dynamic search using a quad-tree [22], (3) breadth-first search on a grid-based graph [26] [27], (4) brute-force SPM [16], (5) our method. The number of obstacles (O) and vertices (V) in the environment are included for the last three methods. Some hardware details were not specified in the papers.

6.1 Benchmarks

We evaluate the performance of our algorithm with several benchmarks where we use a framebuffer resolution of 1000x1000 on a Nvidia GeForce GTX 970 GPU and an Intel Core i7 3.40 GHz computer with 16GB of memory.

Table 1 shows average execution times for computing 100 single-source OPMs with random source points in $\mathcal{D} - \mathcal{O}$. The table shows times both with and without transferring the resulting OPM back to the host memory.

Figure 14 charts out computation times on the Profiling maps. These maps are composed of uniform rows of square obstacles (see Figure 15) with large visible areas from all points in the map. This represents a worst-case scenario for our method because there are large areas visible from all vertices. Still we observe that the increase in computation time is not too distant from linear, given the parallel execution of the GPU rasterization operations.

Table 2 shows that our method is able to compute an OPM and return optimal paths faster than some previous GPU-based methods which perform grid-based search and thus are not optimal for polygonal inputs. For example, Kapadia et al. [21] gives times to plan paths on a grid environment with similar resolution to the buffer used in our benchmarks, 1024x1024, as follows: between 32.931 and 49.126 seconds for a GT 650M and between 21.246 and 30.778 seconds for a GTX 680. While it would be disingenuous to directly compare these numbers to our benchmarks, which used a newer GTX 970, we nevertheless believe that a new card will not offer the significant speed up that would be required to match even the 2.80 second running time we achieved on our most complicated map. In a later work they employed a quad-tree to speed up the computation [22], but sacrificing even more optimality in the process.

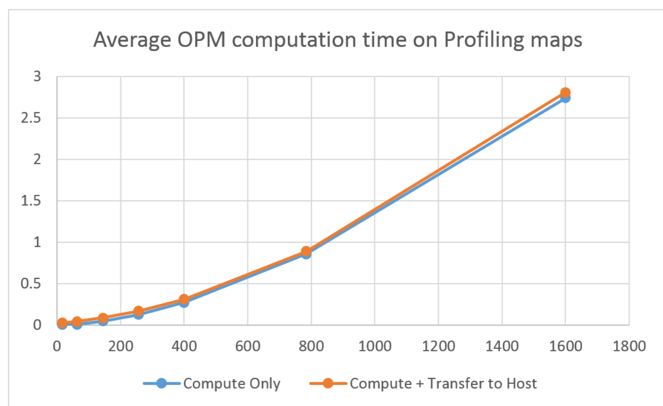


Fig. 14: The x axis represents the number of obstacle vertices in the scene, and the y axis represents the computation time in seconds.

The BFS method included in Table 2 is highly efficient given that it was achieved on a GPU with significantly less cores. The 0.17 time is for running BFS on a 3D grid-based graph of 1M vertices, which was reported as taking $0.678s$ by Luo et al. [27] and then improved by a factor of 4.1 by Merril et al. [26]. These methods are powerful; however, they are limited to BFS on graphs. In addition to generating approximate results in terms of computing Euclidean shortest paths, discrete search alone does not include visibility tests from generators, which are needed to identify the regions of an SPM. Although our method uses a grid of pixels to represent produced OPMs, our search is essentially vertex-based (not pixel-based), allowing us to achieve optimal paths in the Euclidean plane for each pixel.

In addition to the informal comparisons described above, we have also performed benchmarks against the CPU method of Xin and Wang [32] available in CGAL, which is an efficient window propagation method for computing geodesic paths on meshes. Window propagation methods represent a popular approach to generate distance fields and could be extended to build SPMs. Table 3 summarizes our results in terms of path queries per second. For both algorithms, one million points were randomly generated on the map and then used as query points. As can be seen from the table, in all cases tested our OPMs were able to answer a significantly larger number of path queries per second. This follows from the fact that point location is a trivial constant time operation in OPMs, and after that, paths are constructed by simple concatenation of parent points from the query point. The faster query time basically follows from the grid-based representation of our method.

We found that comparing construction times is not straightforward. Our method was slower in computing our per-pixel SPM representation than the time taken by the CGAL method to compute their sequence tree; however, a sequence tree only represents paths to the vertices of the obstacles, and trivially querying the structure to construct path information for every pixel would lead to slower construction times.

6.2 Discussion

Although our method uses a framebuffer grid and thus samples the environment at the level of pixels, the framebuffer is just used as a representation of the produced map. All distances computed during the method execution are calculated exactly using the original coordinates of the sources and obstacle vertices. This

			Queries per second		
Map name	**F**	**AVP**	**CGAL**	**OPM**	**Improv.**
Concave1	18	2.80	1,530,456	9,678,293	6.3x
Concave2	124	3.47	723,589	7,958,298	11.0x
Spiral	42	6.48	1,152,206	5,435,787	4.7x
SpmEx1	23	2.68	1,301,066	9,669,309	7.4x
SpmEx2	119	3.24	720,928	8,278,968	11.5x
Profiling0	26	2.58	1,150,880	10,511,710	9.1x
Profiling1	98	2.95	704,423	9,273,255	13.2x
Profiling2	218	3.29	457,435	8,164,065	17.8x
Profiling3	386	3.55	373,985	7,815,736	20.9x
Profiling4	602	3.90	232,336	7,267,917	31.3x
Profiling5	1178	4.57	153,881	6,075,666	39.5x
Profiling6	2402	5.54	92,909	5,011,099	53.9x

TABLE 3: Number of path queries per second. F is the number of faces (triangles) on each map and AVP is the average number of vertices in the paths computed. The last column shows the improvement obtained with OPMs.

means that there is no accumulation of error introduced by the method when integrating lengths of solution paths.

Only the region borders formed by collision fronts are affected by the pixel approximation since they decide the first parent point to take when starting a shortest path to the closest source. After the first parent point is selected, the next ones are not subject to any approximations as they are determined only from floating point computations with the coordinates of the input vertices. If a query point is less than half of one pixel diagonal length away from a region border, it is possible that its pixel is not associated with the optimal parent. It is however still possible to guarantee global optimality in such cases, by just testing among the neighboring pixels which parent vertex is in fact leading to the shortest path using their exact accumulated distances to the closest source. Cases of multiple distinct shortest paths for a same query point are possible; however, our representation does not provide a direct way to reconstruct all such paths for a query point.

In all cases, a suitable framebuffer grid resolution is expected to be chosen guaranteeing that every grid pixel contains at most one source point or obstacle vertex, and no original free space between obstacles or shadow regions disappear in the grid representation. Under these conditions our method will provide correct minimum shortest paths.

Besides being resolution-sensitive the main limitation of our method is that it may only be suitable for real-time simulations in environments of moderate size. Our method is slower than state-of-the-art path finding solutions that focus on speed of computation instead of global optimality [7]. However, our performance times have the potential to increase over time given the rapid expansion of GPU-based computing hardware and techniques.

7 Conclusions

We have introduced in this paper a novel shader-based GPU method for computing optimal path maps addressing multiple types of sources and weights at vertices representing speed changes. We also uncover the interesting property that speed changes may lead to maps with disconnected regions associated to a same parent generator, something that cannot happen in traditional SPMs. The achieved capabilities have clear practical applications and were not explored before in an optimal way. Our benchmarks show that our method outperforms comparable approaches in many cases.

Our approach opens new directions for incorporating navigation mapping techniques within the graphics pipeline. Our maps can instantly guide agents in multi-agent simulations from GPU buffers storing distances to the closest target and the next point to aim for from any position in the environment.

Acknowledgments

This research was sponsored by the Army Research Office and was accomplished under Grant Number W911NF-17-1-0463. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] M. De Berg, O. Cheong, M. Van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Application*. Springer, 2008.

[2] C. Camporesi and M. Kallmann, “Computing Shortest Path Maps with GPU Shaders,” in *Proceedings of the Seventh International Conference on Motion in Games*, ser. MIG ’14. New York, NY, USA: ACM, 2014, pp. 97–102.

[3] M. Likhachev, D. Ferguson, G. Gordon, A. T. Stentz, and S. Thrun, “Anytime Dynamic A*: An Anytime, Replanning Algorithm,” in *Proceedings of the International Conference on Automated Planning and Scheduling*, June 2005.

[4] A. Nash, K. Daniel, S. Koenig, and A. Feiner, “Theta*: Any-angle Path Planning on Grids,” in *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*, ser. AAAI’07. AAAI Press, 2007, pp. 1177–1183.

[5] R. Geraerts, “Planning Short Paths with Clearance using Explicit Corridors,” in *IEEE International Conference on Robotics and Automation*, 2010, pp. 1997–2004.

[6] R. Oliva and N. Pelechano, “NEOGEN: Near Optimal Generator of Navigation Meshes for 3D Multi-Layered Environments,” *Computers & Graphics*, vol. 37, no. 5, pp. 403–412, Aug. 2013.

[7] M. Kallmann, “Dynamic and Robust Local Clearance Triangulations,” *ACM Transactions on Graphics*, vol. 33, no. 4, 2014.

[8] E. Welzl, “Constructing the Visibility Graph for n-line Segments in O(n^2) Time,” *Information Processing Letters*, vol. 20, no. 4, 1985.

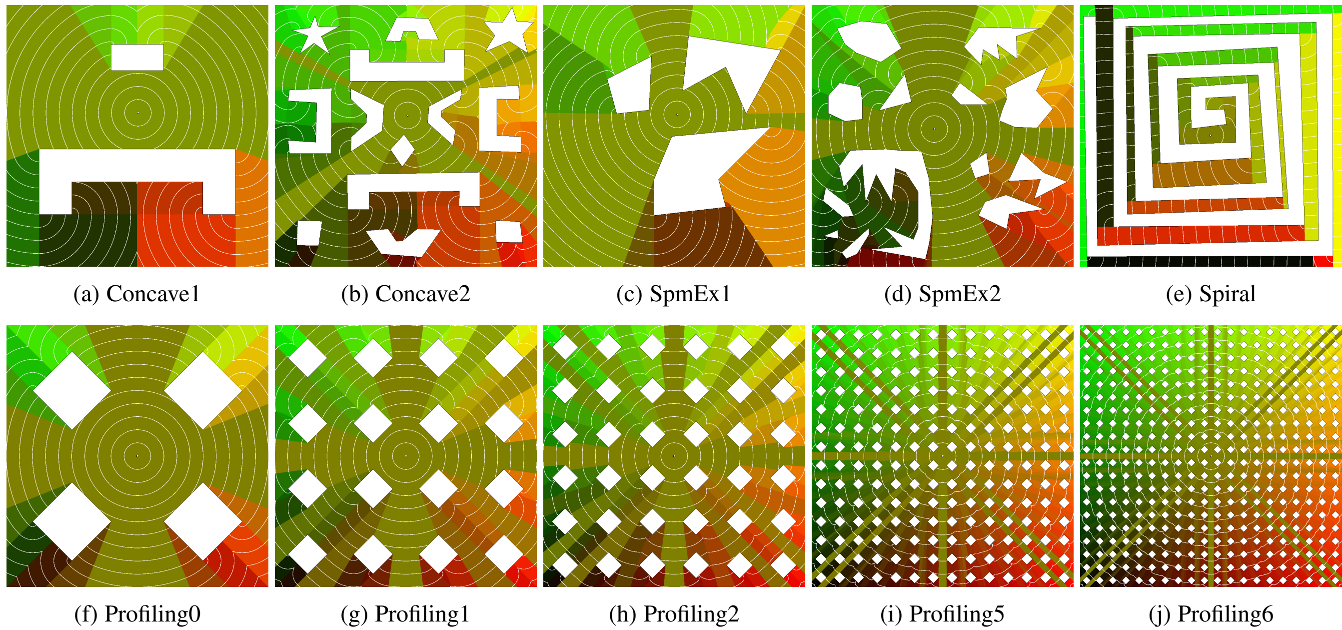
[9] M. H. Overmars and E. Welzl, “New Methods for Computing Visibility Graphs,” in *Proceedings of the Fourth Annual Symposium on Computational Geometry*, ser. SCG ’88. New York, NY, USA: ACM, 1988, pp. 164–171.

[10] J. A. Storer and J. H. Reif, “Shortest Paths in the Plane with Polygonal Obstacles,” *Journal of the ACM*, vol. 41, no. 5, pp. 982–1012, Sep. 1994.

[11] N. J. Nilsson, “A Mobius Automation: An Application of Artificial Intelligence Techniques,” in *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, ser. IJCAI ’69. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1969, pp. 509–520.

[12] D. T. Lee and F. P. Preparata, “Euclidean Shortest Paths in the Presence of Rectilinear Barriers,” *Networks*, vol. 14, no. 3, pp. 393–410, 1984.

[13] J. S. B. Mitchell, “A New Algorithm for Shortest Paths Among Obstacles in the Plane,” *Annals of Mathematics and Artificial Intelligence*, vol. 3, no. 1, pp. 83–105, 1991.



(a) Concave1 (b) Concave2 (c) SpmEx1 (d) SpmEx2 (e) Spiral

(f) Profiling0 (g) Profiling1 (h) Profiling2 (i) Profiling5 (j) Profiling6

Fig. 15: Single-source OPM results.

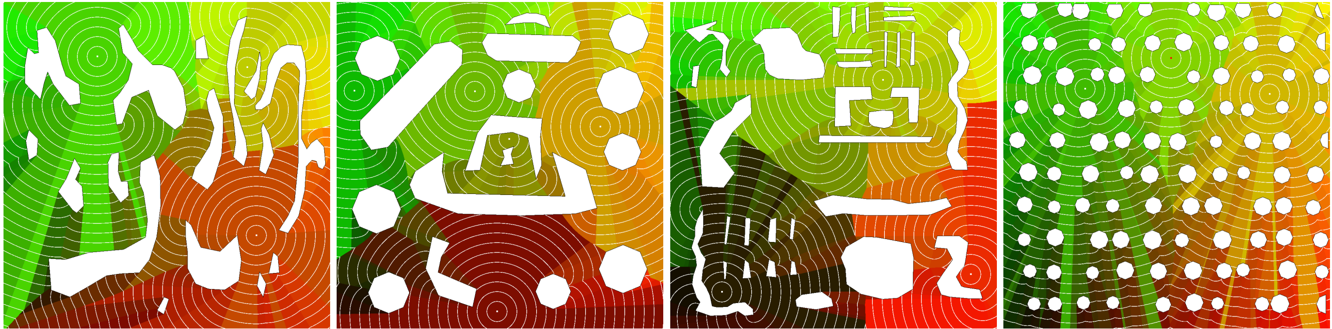


Fig. 16: Multiple-source OPM results.

[14] ——, “Shortest Paths Among Obstacles in the Plane,” in *Proceedings of the Ninth Annual Symposium on Computational Geometry*, ser. SCG ’93. New York, NY, USA: ACM, 1993, pp. 308–317.

[15] J. Hershberger and S. Suri, “An Optimal Algorithm for Euclidean Shortest Paths in the Plane,” *SIAM Journal on Computing*, vol. 28, pp. 2215–2256, 1999.

[16] E. Wynters, “Constructing Shortest Path Maps in Parallel on GPUs,” *Proceedings of 28th Annual Spring Conference of the Pennsylvania Computer and Information Science Educators*, 2013.

[17] X. Ying, S.-Q. Xin, and Y. He, “Parallel Chen-Han (PCH) Algorithm for Discrete Geodesics,” *ACM Transactions on Graphics*, vol. 33, no. 1, pp. 9:1–9:11, Feb. 2014.

[18] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, and R. Kimmel, “Parallel Algorithms for Approximation of Distance Maps on Parametric Surfaces,” *ACM Transactions on Graphics*, vol. 27, no. 4, pp. 104:1–104:16, Nov. 2008.

[19] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver, “Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware,” in *Proceedings of SIGGRAPH*, ser. SIGGRAPH ’99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286.

[20] M. Henderson, J. T. Kider, M. Likhachev, and A. Safonova, “High-Dimensional Planning on the GPU,” in *IEEE International Conference on Robotics and Automation*, May 2010, pp. 2515–2522.

[21] M. Kapadia, F. M. Garcia, C. D. Boatright, and N. I. Badler, “Dynamic Search on the GPU,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nov 2013, pp. 3332–3337.

[22] 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*, May 2014, pp. 1631–1638.

[23] M. Qi, T.-T. Cao, and T.-S. Tan, “Computing 2D Constrained Delaunay Triangulation Using the GPU,” in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D ’12. New York, NY, USA: ACM, 2012, pp. 39–46.

[24] S. J. Guy, J. Chhugani, C. Kim, N. Satish, M. C. Lin, D. Manocha, and P. Dubey, “ClearPath: Highly Parallel Collision Avoidance for Multi-Agent Simulation,” in *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, 2009, pp. 177–187.

[25] D. Delling, A. Goldberg, A. Nowatzyk, and R. Werneck, “PHAST: Hardware-Accelerated Shortest Path Trees,” Tech. Rep., September 2010.

[26] D. Merrill, M. Garland, and A. Grimshaw, “Scalable GPU Graph Traversal,” in *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’12. New York, NY, USA: ACM, 2012, pp. 117–128.

[27] L. Luo, M. Wong, and W.-m. Hwu, “An Effective GPU Implementation of Breadth-first Search,” in *Proceedings of the 47th Design Automation Conference*, ser. DAC ’10. New York, NY, USA: ACM, 2010, pp. 52–55.

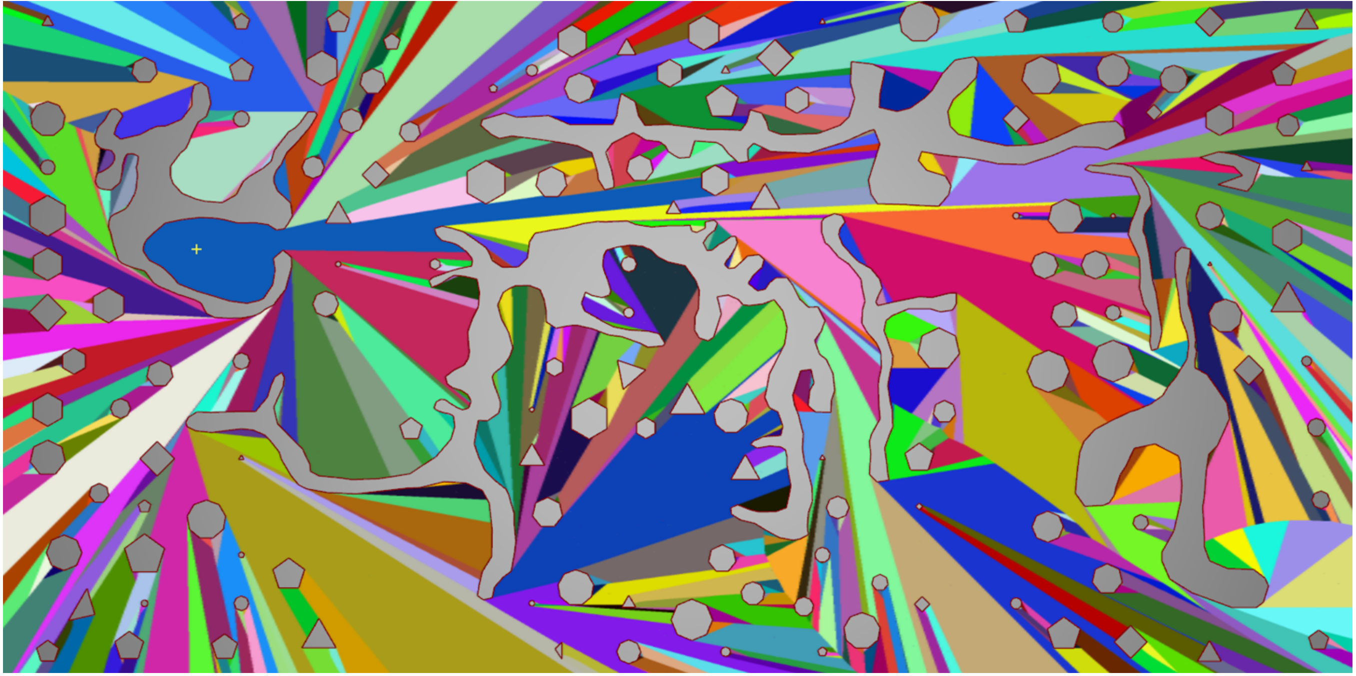


Fig. 17: Example illustrating the interesting boundaries obtained in a complex SPM example. This SPM has a single source point marked with a cross on the left side of the diagram.

[28] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou, "The Discrete Geodesic Problem," *SIAM Journal on Computing*, vol. 16, no. 4, pp. 647–668, Aug. 1987.

[29] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. J. Gortler, and H. Hoppe, "Fast Exact and Approximate Geodesics on Meshes," in *Proceedings of SIGGRAPH*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005, pp. 553–560.

[30] D. Bommes and L. Kobbelt, "Accurate Computation of Geodesic Distance Fields for Polygonal Curves on Triangle Meshes," in *Proceedings of the Vision, Modeling, and Visualization Conference*, 01 2007, pp. 151–160.

[31] J. Chen and Y. Han, "Shortest Paths on a Polyhedron," in *Proceedings of the Sixth Annual Symposium on Computational Geometry*, ser. SCG '90. New York, NY, USA: ACM, 1990, pp. 360–369.

[32] S.-Q. Xin and G.-J. Wang, "Improving Chen and Han's Algorithm on the Discrete Geodesic Problem," *ACM Transactions on Graphics*, vol. 28, no. 4, pp. 104:1–104:8, Sep. 2009.

[33] Y. Qin, X. Han, H. Yu, Y. Yu, and J. Zhang, "Fast and Exact Discrete Geodesic Computation Based on Triangle-oriented Wavefront Propagation," *ACM Transactions on Graphics*, vol. 35, no. 4, pp. 125:1–125:13, Jul. 2016.

[34] G. Peyré and L. Cohen, "Geodesic Remeshing Using Front Propagation," *International Journal of Computer Vision*, p. 156, 2006.

[35] M. Novotni and R. Klein, "Computing Geodesic Distances on Triangular Meshes," *10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2002.

[36] J. Tang, G.-S. Wu, F.-Y. Zhang, and M.-M. Zhang, "Fast Approximate Geodesic Paths on Triangle Mesh," *International Journal of Automation and Computing*, vol. 4, pp. 8–13, 01 2007.

[37] Y. Qin, H. Yu, and J. Zhang, "Fast and Memory-Efficient Voronoi Diagram Construction on Triangle Meshes," *Computer Graphics Forum*, vol. 36, pp. 93–104, Aug. 2017.

[38] L. Lu, B. Lévy, and W. Wang, "Centroidal Voronoi Tessellation of Line Segments and Graphs," *Computer Graphics Forum*, vol. 31, no. 2pt4, pp. 775–784, 2012.

[39] C. Xu, Y.-J. Liu, Q. Sun, J. Li, and Y. He, "Polyline-sourced Geodesic Voronoi Diagrams on Triangle Meshes," *Computer Graphics Forum*, vol. 33, no. 7, pp. 161–170, Oct. 2014.

Renato Farias received a BSc degree in Computer Science in 2011 from the Federal University Fluminense, Brazil, and a MSc degree in the area of Computer Graphics from the Federal University of Rio de Janeiro in 2014. He is currently working toward his PhD degree at the University of California, Merced, on the topic of GPU-based methods and representations for navigation. His research interests include computer graphics, path planning, and GPU programming.

Marcelo Kallmann is founding faculty and full professor of computer science at the University of California, Merced. He holds a PhD from the Swiss Federal Institute of Technology in Lausanne (EPFL), and before joining UC Merced he was a research faculty member at the University of Southern California (USC) and a scientist at the USC Institute of Creative Technologies. His areas of research include motion planning, computer animation and virtual reality. He has been on the program committee of major conferences in computer graphics and robotics, has served as program co-chair for the 5th International Conference on Motion in Games (2012), associate editor for ICRA (2015 and 2016), and associated editor for the Computer Animation and Virtual Worlds journal (2018-2019). His research has been supported by the US National Science Foundation and the Army Research Office, and his work on triangulations for path planning runs inside The Sims 4, the best selling PC game of 2014 and 2015.