In Proceedings of Motion in Games (MIG), Los Angeles, 2014. This version is the original manuscript of the authors.

Computing Shortest Path Maps with GPU Shaders

Carlo Camporesi*
University of California, Merced

Marcelo Kallmann†
University of California, Merced

Abstract

We present in this paper a new GPU-based approach to compute Shortest Path Maps (SPMs) from a source point in a polygonal domain. Our method takes advantage of GPU polygon rasterization with shader programming. After encoding the SPM in the frame buffer, globally shortest paths are efficiently computed in time proportional to the number of vertices in the path, and length queries are computed in constant time. We have evaluated our method in multiple environments and our results show a significant speedup in comparison to previous approaches.

CR Categories: I.3.7 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric Algorithms, languages, and systems.

Keywords: Path Planning, Shortest Paths, the Shortest Path Map.

1 Introduction

The computation of Euclidean Shortest Paths among obstacles is a classical problem with applications in many areas. The Shortest Path Map (SPM) [Lee and Preparata 1984; Mitchell 1993] is a structure that efficiently captures all the shortest paths in a polygonal domain, with respect to a given source point. While SPMs are well-known in computational geometry, they are still little known in other areas of computer graphics and their use in practical applications has not yet been much explored. One reason for this situation is that efficient methods for computing SPMs are not straightforward to be implemented, what justifies alternative GPU-based approaches to be developed.

We introduce in this paper a new method for computing SPMs based on GPU shader programming techniques. Our experiments show that our method is significantly faster than previously reported results. Our method first relies on standard CPU algorithms for computing the shortest path tree of the obstacle set, and then applies the proposed shaders to encode the SPM in the frame buffer with arbitrary resolution. This last step is performed entirely with GPU operations. See Figure 1 for examples.

Given a polygonal environment of n vertices, our discrete (pixel-based) SPM representation is then able to compute shortest paths from the source point to any reachable point in the environment in $O(k)$, where k is the number of turns in the path; and to compute shortest path length queries in constant time. In contrast, continuous approaches for computing the SPM will answer these queries in $O(\log(n) + k)$ and $O(\log(n))$ time respectively.

*e-mail:ccamporesi@ucmerced.edu
†e-mail:mkallmann@ucmerced.edu

Figure 1: *Example of Results. Left: example SPMs generated from the source point marked with a small yellow cross. Each region in the SPM has a single generator, which makes the reconstruction of shortest paths very efficient. The red path is the shortest path to an arbitrary target point. Right: geodesic flood-fill expansion from the source point. The colors, from blue to dark red, represent the distance to the source point, from close to far.*

2 Related Work

The most common approach to solve Euclidean shortest path problems is to first build the visibility graph [Nilsson 1969; De Berg et al. 2008] of the environment and then run standard graph search algorithms on the graph. The visibility graph however is a graph of $O(n^2)$ nodes, where n is the total number of vertices in the environment [Welzl 1985; Storer and Reif 1994; Overmars and Welzl 1988]; and in addition, a new graph search is needed for each given shortest path query.

Efficient approaches to the problem are based on the continuous Dijkstra paradigm [Lee and Preparata 1984], which simulates the propagation of a continuous wavefront maintaining equal length to the source point. When applied to the whole environment the result is the Shortest Path Map (SPM), which is able to efficiently answer shortest path queries with respect to the source point. Efficient algorithms exist to build SPMs [Mitchell 1993; Hershberger and Suri 1997]; however, they involve techniques and data structures not practical to be implemented.

Alternative to continuous approaches, a discrete pixel-based method has already been developed taking advantage of the massive parallelization of GPUs. Like our approach, Wynters [2013] first computes the visibility graph and the shortest path tree (SPT) of the environment to then build the SPM. The SPM generation however does not rely on rasterization with shaders like in our ap-

proach, and instead is based on a brute-force parallelized per-pixel determination of the closest SPT point, leading to the subdivision of the pixel space in SPM regions. The results obtained with our proposed method demonstrate a significant speed-up, of about 20 times, in similar reported conditions [Wynters 2013]. We achieve a level of complexity and detail not seen before in previous work.

With respect to previous work in shader programming techniques, our work mainly relies on the use of shadow volumes [Eisemann et al. 2009]. Algorithms for rendering hard shadows in real time can be roughly sorted into three categories: shadow mapping [Williams 1978; Lauritzen et al. 2011; Lloyd et al. 2008], alias-free shadow maps [Aila and Laine 2004; Johnson et al. 2005; Sintorn et al. 2011], and shadow volumes [Fuchs et al. 1985; Heidmann 1991]. Our work relies on the generation of shadow volumes with a variation of the *z-fail* approach [Carmack 2000; Bilodeau and Songy 1999; Everitt and Kilgard 2002]. This approach is known to be slower than methods using shadow maps, due to higher overdraw and the need for near and far clipping geometry [Laine 2005]; however, it does not suffer from (quite severe) resolution aliasing artifacts, and it is not camera-dependent, allowing in a single pass to generate shadows in every scene direction.

3 Method

Our overall method consists of three main steps: environment pre-computation, visibility graph and SPT computation, and SPM generation and query, as summarized in Algorithm 1. See also Figure 2.

Algorithm 1 Main procedure including pre-computation stages.

```
1: procedure GENERATESPM
2:     e ← environment
3:     p ← source point
4:     if (new environment) then
5:         genEnvVolume(e)                    ▷ Section 3.1
6:         vg ← genVisGraph(e)                ▷ Section 3.2
7:     else
8:         vg ← current visibility graph
9:     genSPM(p, vg, e)                       ▷ Section 3.3
```

Each input polygonal obstacle is identified by a unique ID (pid). We assume that obstacles are disjoint, represented by vertices in counter clockwise order, and that they have at least 3 vertices.

Figure 2: *After the SPT is computed from the visibility graph (left), visible regions for each vertex are determined (center) and composed following the SPT ordering to form the SPM (right).*

3.1 Environment Pre-Computation

The first step of our algorithm is to generate extruded volumes of the input polygons along the z-axis of the environment, which is the orthogonal axis to the plane of the polygonal domain. See Figure 3. Extruded polygons are represented as triangle meshes and they are enhanced with triangle adjacency information [Shreiner et al. 2013] in order to enable the generation of shadow volumes from any point of view through geometry shaders.

3.2 Visibility Graph

Our generation of the visibility graph follows the standard approach of connecting all pairs of vertices that are visible to each other, with the optimization of only including edges that connect two convex vertices. Our implementation determines visible vertices by traversing a triangulation of the environment outwards from each vertex. The method takes $O(n^3)$ time however in practice it efficiently culls non-visible regions of the environment. Each node of our visibility graph stores a list with the ids of all visible obstacles.

3.3 SPM generation

Each time a new source point is given, the point is first connected with edges to its visible nodes of the visibility graph. An exhaustive Dijkstra expansion starting from the newly inserted source node is then executed and the resulting expansion tree is the SPT of the polygonal environment. Each node of the SPT will correspond to one specific obstacle vertex. See Figure 2-left.

During the SPT construction process each vertex is labeled to identify if the edges connected to the vertex are both visible (partially or entirely) from its SPT parent vertex, or if one of them is completely occluded. SPT nodes adjacent to visible edges are removed from the SPT because they will not be generators of a SPM region [Mitchell 1993]. The remaining nodes will be generators and a unique color ID is assigned to them. The generator nodes are also stored in a list sorted by the Euclidean geodesic distance of each node to the source point.

Our approach for obtaining the SPM is inspired by the GPU method of rendering cones as a way to represent distance to boundaries, originally proposed to compute generalized Voronoi diagrams [Hoff et al. 1999]. In our SPM case we use clipped 3D cones to represent wavefront expansions from generator vertices of the SPT. Each cone has its height equal to its radius. In the source point case the cone does not need to be clipped. When cones are rendered from an orthographic point of view the generated depth buffer will contain distances to the generator point.

Cones are placed along the Z axis at different heights, according to the geodesic distance from their generators to the source point. See Figure 3-b. Using these settings, if two intersecting cones (with different color IDs) are rendered the Z-buffer depth test will only keep, for each Z-buffer pixel, the depth value of the closest cone geometry to the camera's point of view. Each value in the Z-buffer is still linked to the cone's geometry that generated it, so that the cone's color can be assigned to its specific pixel in the frame buffer during the rendering step. The final image produced consists of colored regions where the color of each pixel is an ID that directly gives access to its closest SPT generator node, and the depth value in the Z-buffer contains, for each pixel, the geodesic distance to the source point of the SPM.

One important observation is that it is not enough to simply place a cone at its apex generator vertex at the correct height because it may intersect with unrelated cones placed in the environment, leading to wrong results. To address this issue we have to ensure that each cone propagation will not affect regions that are not visible from the generator vertex of the cone. This process could be addressed by generating clipped cones per vertex, however imposing CPU computation of specific shapes per vertex, which would have to be computed and passed to the graphics hardware for each new SPM source.

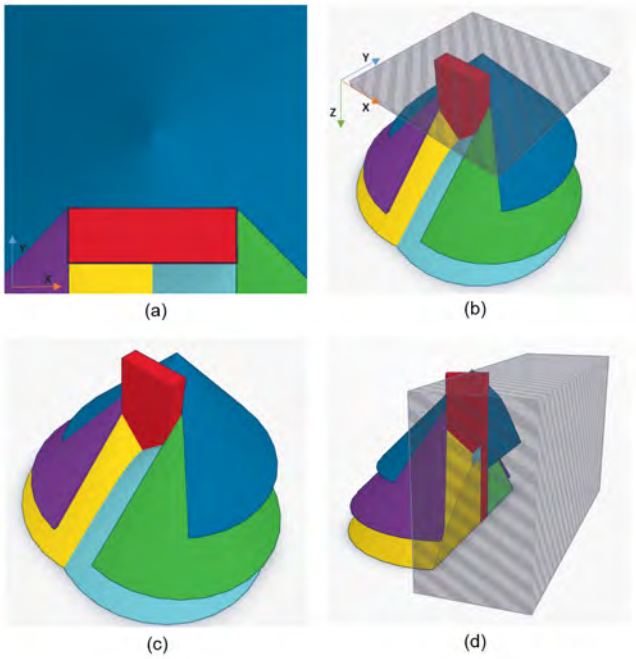

Figure 3: *Cone placement and obstacle extrusion. (a) Cones rendered with flat shading and projected to the obstacle plane; the red rectangle is the obstacle. (b) Perspective view of the clipped cones and the extruded obstacle. Cones are clipped against their visibility regions. The gray plane is the obstacle plane. (c) Perspective view without the obstacle plane. (b) Sliced view of the clipped cones showing the internal part of the extruded polygon. The gray box shows the clipping volume.*

Our approach makes use of the concept of point light propagation and shadow. We consider each SPT node to be an omni-directional light source and we generate directly in GPU shadow volumes that will be used to identify, during each cone's rendering, which specific portions of the cone should be used for rendering and therefore for updating only the relevant color and depth buffer values. We have used an optimized version of the *z-fail* shadow volume algorithm to generate shadowed regions in the scene to be used for cone clipping. Figure 2-center exemplifies the visible region generated by the shadow volume algorithm and applied to the cone of a source point. Figure 3 shows multiple views of the SPM generation from several cones placed and clipped by our method.

This approach relies on sequentially updating the buffers and in our setup it is required that the cone rendering follows the strict order given by its sorted geodesic distance from the SPM source. The sorted list of vertices is created during the SPT computation step. Farthest cones have to be rendered first.

Cone rendering has been implemented in two passes. The first pass updates the stencil buffer with the shadow volume information generated from each light source (a SPT generator vertex) against the polygonal obstacles extrusion geometry. This stage is also optimized to pre-cull occluded polygons using the pid defined in section 3.1. The second pass consists of the actual rendering of the cone's geometry with the stencil operation enabled. A cone is rendered only where its shadow volume was not present in the scene. The overall method is detailed in Algorithm 2.

We use two shader programs. The first shader technique (*techFlat*) simply renders the cones with the assigned color without applying any shading information from the geometry. The second shader program (*techShad*) is responsible to compute the shadow volume projecting the environment geometry silhouette to infinite (the projection point is a SPT generator node or the SPM source point). This shader is a modified version of the most common silhouette generation algorithm. The algorithm presents a generic method to create shadows from any point of view using geometry with adjacency information. We have also developed a specialized version from the orthogonal camera point of view using only generic geometry (without adjacency information).

To approximate a vertex projection to infinite, the shader employs the simple solution of using an ϵ value that is very close to 0 to be the w homogeneous component of the normalized direction defined by the light source and the vertex. An alternative technique that does not depend on an ϵ is also possible to be employed here [Everitt and Kilgard 2002].

Algorithm 2 Pseudocode of the SPM rendering routine. The pseudocode below draws to the frame buffer but the actual implementation is extended to render in a frame buffer object.

```
1:  procedure DrawSPM
2:      techShad ← shadow volume technique
3:      techFlat ← flat color technique
4:      polyVol ← extruded obstacles          ▷ details in section 3.1
5:      nlist ← SPT nodes                     ▷ nodes are sorted by distance in reverse order.
6:      V_P ← view * projection matrix
7:  begin:
8:      glEnable(Stencil, Depth, DepthClamp)
9:      glDepthFunc(LeEqual)
10:     for all n : nlist do
11: pass1:                                    ▷ updates the stencil buffer
12:         glClear(StencilBufferBit)
13:         glDrawBuffer(None)
14:         glColorMask(False)
15:         glDepthMask(False)
16:         glDisable(CullFace)
17:         glStencilFunc(Always, 0, 0xFF)
18:         glStencilOpSeparate(Back, Keep, IncrWrap, Keep)
19:         glStencilOpSeparate(Front, Keep, DecrWrap, Keep)
20:         glColorMask(False)
21:         p ← n.position                    ▷ the node position already includes the correct height placement
22:         M ← polyVol.M                     ▷ poly extrusion's model matrix modified so that the volume's height is scaled to a dimension double than the cone's height and the center of the volume is placed at the same height of the light source
23:         techShad.set(M, V_P, p)
24:         polyVol.draw()
25:         glEnable(CullFace)
26:         glDrawBuffer(Back)
27: pass2:                                    ▷ updates the back buffer
28:         glColorMask(True)
29:         glDepthMask(True)
30:         glStencilOpSeparate(Back, Keep, Keep, Keep)
31:         glStencilFunc(Equal, 0, 0xFF)
32:         M ← n.coneInstance.M              ▷ model matrix to place the cone instance in the scene
33:         techFlat.set(M, V_P)
34:         n.coneInstance.draw()
```

Four additional considerations have to be taken into account, as explained below.

Graphics Card Numerical Precision The light source position passed to the shader is often an SPT node (thus exactly at a polygon vertex), and due to precision errors, it can be incorrectly considered by the graphics card to be inside a polygon and in such cases incorrect results may be generated. We avoid this issue by slightly placing the source light outside the obstacle along the bisector vector between the two adjacent edges to the vertex.

Z-Buffer Precision The proposed algorithm relies on the depth buffer precision, which directly depends on the $zNear$ and $zFar$ values. To take full advantage of the depth buffer precision we adaptively compute the $zFar$ value according to the maximum depth dimension needed by the algorithm. We also appropriately set the $zNear$ value considering the camera point of view to correctly render the entire scene when rendering the scene from a generic point of view for debugging purposes.

Z-Fighting Since each cone mesh has the same slope and the cone placement of children expansions are exactly at the parent's frontier, polygons belonging to different cones but with a parent-children relationship may generate Z-fighting artifacts. To avoid this issue we add to the children's cone height a tiny gap. This gap ensures a clipped cone mesh to be always rendered underneath the parent's clipped cone mesh. This value is also adaptively computed since it depends on the depth buffer precision.

Cone Mesh Resolution For correct results the resolution of the cone meshes need to be appropriately set. Adaptive resolutions are also useful to optimize the results [Hoff et al. 1999].

4 Results

We have implemented our shaders in OpenGL 3.3 or above and with C++ for the CPU portion. Our performance evaluations used a GeForce 570 GTX GPU and a single Intel Core i7-2600K CPU at 3.40 GHz. For the frame buffer object render pass we used a 32-bit RGBA render target (to store colors) and a 24-bit depth and 8-bit stencil render target (to store the depth and stencil information respectively). The test resolutions of the render targets (FBO, depth and stencil buffers) were set at 512x512, 1024x1024 and 2048x2048 pixels. We also varied the resolution of the cones to 512, 1024, 1598, 2048, 4196 triangles per cone.

We have evaluated 7 maps with a varying number of square obstacles, and using the center point as the SPM source. We have also tested a generic environment with complex obstacles (map 8). The maps and their SPMs are shown in Figure 5. Table 1 summarizes the statistics. (For additional results see accompanying video available at *http://graphics.ucmerced.edu/publications.html*.)

Table 1: *Evaluated environments. The number of vertices in the input polygon set is n, and n_f refers to the number of triangles representing all extruded volumes from the polygonal obstacles. Time t_{pr} is the environment pre-computation time, which includes the computation of the visibility graph. Column t_{spt} shows the computation time taken to generate the SPT in each given environment.*

map	name	obsts	n	n_f	cones	t_{pr} (ms)	t_{spt} (ms)
1	*2x2*	4	16	48	17	0.12	0.03
2	*4x4*	16	64	192	53	0.4	0.12
3	*6x6*	36	144	432	109	6.4	0.28
4	*8x8*	64	256	768	197	3.19	0.63
5	*10x10*	100	400	1200	307	24.23	1.18
6	*14x14*	198	784	2352	577	88.97	3.55
7	*20x20*	400	1600	4800	1207	163.6	12.2
8	*Gen.*	144	1647	6012	761	60.76	3.25

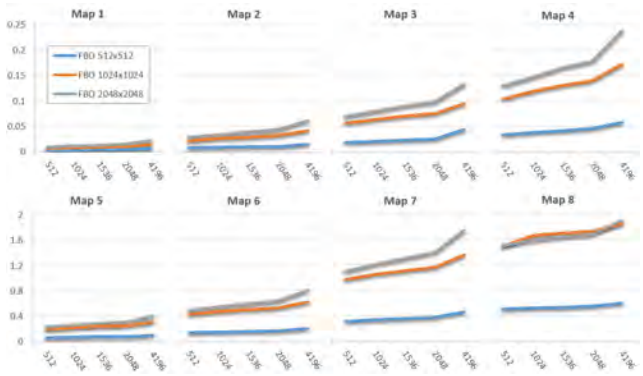

Figure 4: *Computation times taken to run our SPM shaders in several environments and resolutions. The reported times are in seconds. The horizontal axis in each graph represents the number of triangles used to render each cone mesh.*

In comparison to previous results [Wynters 2013], our approach shows to be over 20 times faster, for a same number of obstacles, map resolution and a slightly less powerful graphics card (480 computational cores instead of 512 cores). Figure 4 shows the time (in seconds) to generate the SPMs in different conditions. Each time has been calculated after flushing the graphics card rendering pipeline with *glFinish*. In normal settings this command is not necessary and the rendering time for querying will be even smaller.

As in Wynters [2013], our SPM representation supports constant time distance queries for any discrete point in the SPM, and paths can be returned in time proportional to the number of turns along the path. To test this we have developed a geodesic particle simulation environment where particles efficiently travel along their shortest paths to the source point. See Figure 6.

We have also developed a shader that simulates the continuous geodesic flood-fill expansion using a color map representation. The process replaces the *techFlat* shader by the geodesic flood-fill program, which translates the geodesic distance of each pixel to a specific color value considering its shortest path to the source point. The shader uses HSV color representation to uniformly map distances to colors and white spaces are added to highlight the boundaries of the propagated waves. Results can be seen in Figure 1-right and Figure 7.

5 Conclusion and Future Work

We have introduced a novel approach to generate discrete Shortest Path Maps from polygonal domains using GPU shader programming. The proposed algorithm has shown to be over 20 times faster than previously reported results. This improvement has allowed us to generate more complex results, and to obtain images of SPMs with complexity and detail not shown before in previous work. Our results help to provide a deeper understanding of SPMs and their applications.

We believe that many improvements to the presented shader programs can be achieved and many extensions to the proposed algorithms are possible. For instance, Pixel Buffer Objects instead of Frame Buffer Objects may result in faster non-blocking SPM queries, and stencil-based 2D shadow projection using quads extruded from visible edges instead of full 3D shadows may also generate faster results.

Acknowledgments We would like to thank Joseph S. B. Mitchell for fruitful discussions on this work.

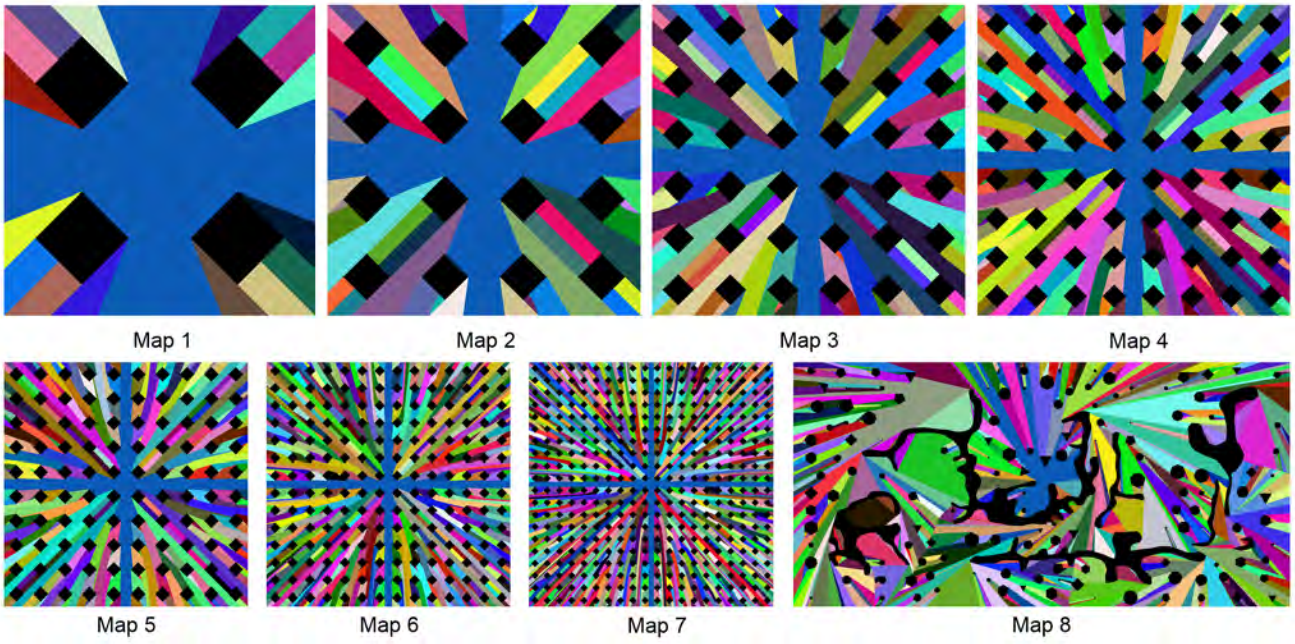

Figure 5: *Environments used in the reported performance evaluations. Each image shows the polygonal domain and its resulting SPM.*

Figure 6: *Geodesic particle simulation: each particle, randomly placed in the scene, is traveling to the source point along its shortest path.*

References

AILA, T., AND LAINE, S. 2004. Alias-free shadow maps. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, EGSR '04, 161–166.

BILODEAU, W., AND SONGY, M., 1999. Real time shadows.

CARMACK, J., 2000. Z-fail shadow volumes. Internet Forum.

DE BERG, M., CHEONG, O., AND VAN KREVELED, M. 2008. *Computational geometry: algorithms and application*. Springer.

EISEMANN, E., ASSARSSON, U., SCHWARZ, M., AND WIMMER, M. 2009. Casting shadows in real time. In *ACM SIGGRAPH ASIA '09 Courses*, ACM, New York, NY, USA.

EVERITT, C., AND KILGARD, M. J., 2002. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. Published online at http://developer.nvidia.com.

FUCHS, H., GOLDFEATHER, J., HULTQUIST, J. P., SPACH, S., AUSTIN, J. D., BROOKS, JR., F. P., EYLES, J. G., AND POULTON, J. 1985. Fast spheres, shadows, textures, transparencies, and imgage enhancements in pixel-planes. In *Proceedings of SIGGRAPH '85*, ACM, New York, NY, USA, 111–120.

HEIDMANN, T. 1991. *Real shadows, real time*. Iris Universe 18. Silicon Graphics, Inc.

HERSHBERGER, J., AND SURI, S. 1997. An optimal algorithm for euclidean shortest paths in the plane. *SIAM J. Comput 28*, 2215–2256.

HOFF, III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH '99*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 277–286.

JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph. 24*, 4 (Oct.), 1462–1482.

LAINE, S. 2005. Split-plane shadow volumes. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ACM, New York, NY, USA, HWWS '05, 23–32.

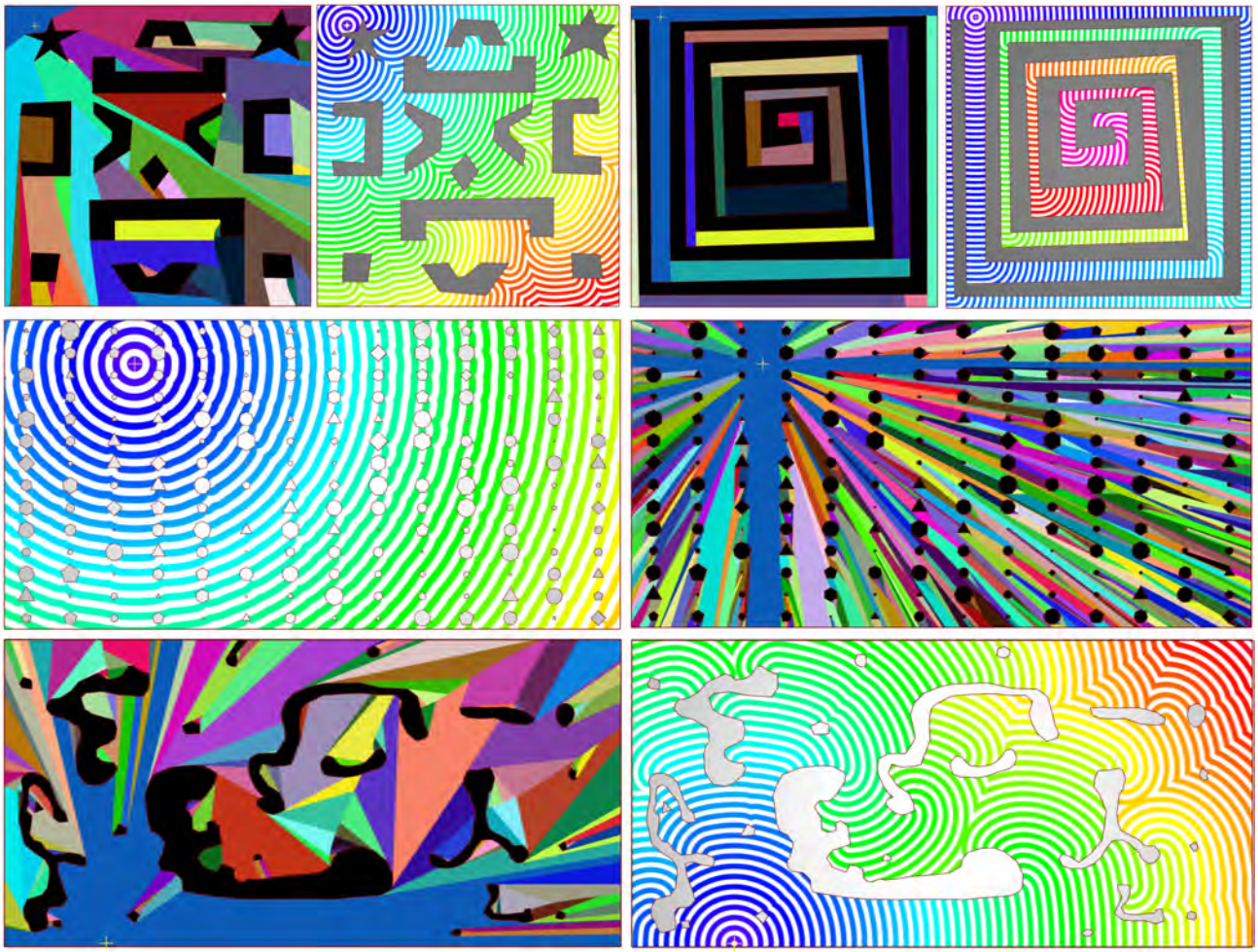

Figure 7: *Additional environments with their SPMs and geodesic flood-fill. Curved boundaries in an SPM diagram are hyperbolic arcs defined by front collisions. These boundaries are here obtained in the frame buffer by Z-Buffer occlusion determination of intersecting cones.*

LAURITZEN, A., SALVI, M., AND LEFOHN, A. 2011. Sample distribution shadow maps. In *Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '11, 97–102.

LEE, D. T., AND PREPARATA, F. P. 1984. Euclidean shortest paths in the presence of rectilinear barriers. *Networks 14*, 3, 393–410.

LLOYD, D. B., GOVINDARAJU, N. K., QUAMMEN, C., MOLNAR, S. E., AND MANOCHA, D. 2008. Logarithmic perspective shadow maps. *ACM Trans. Graph. 27*, 4 (Nov.), 106:1–106:32.

MITCHELL, J. S. B. 1993. Shortest paths among obstacles in the plane. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, ACM, New York, NY, USA, SCG '93, 308–317.

NILSSON, N. J. 1969. A mobius automation: An application of artificial intelligence techniques. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, IJCAI '69, 509–520.

OVERMARS, M. H., AND WELZL, E. 1988. New methods for computing visibility graphs. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, ACM, New York, NY, USA, SCG '88, 164–171.

SHREINER, D., SELLERS, G., LICEA-KANE, B., AND KESSENICH, J. M. 2013. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 4.1*. Graphics programming. Addison Wesley Professional.

SINTORN, E., OLSSON, O., AND ASSARSSON, U. 2011. An efficient alias-free shadow algorithm for opaque and transparent objects using per-triangle shadow volumes. *ACM Trans. Graph. 30*, 6 (Dec.), 153:1–153:10.

STORER, J. A., AND REIF, J. H. 1994. Shortest paths in the plane with polygonal obstacles. *J. ACM 41*, 5 (Sept.), 982–1012.

WELZL, E. 1985. Constructing the visibility graph for n-line segments in $O(n^2)$ time. *Information Processing Letters 20*, 4.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *Proceedings of SIGGRAPH'78 12*, 3, 270–274.

WYNTERS, E. 2013. Constructing shortest path maps in parallel on GPUs. *Proceedings of 28th Annual Spring Conference of the Pennsylvania Computer and Information Science Educators*.