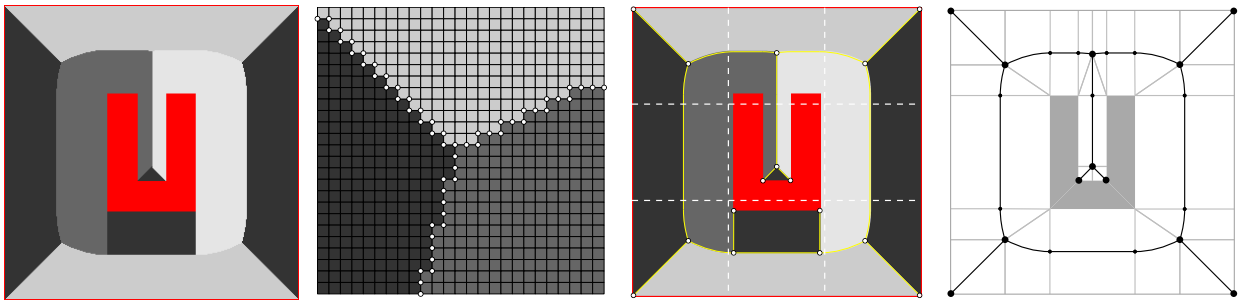
GPGPU-Accelerated Construction of High-Resolution Generalized Voronoi Diagrams and Navigation Meshes

Rudi Bonfiglioli* Textkernel
Wouter van Toll† Utrecht University
Roland Geraerts‡ Utrecht University



(a) *Approximated GVD* **(b)** *Filtered grid* **(c)** *Multiple tiles* **(d)** *ECM navigation mesh*

Figure 1: *The pipeline of our algorithm. (a) For a set of obstacles (a 'U' and a bounding box), we approximate the Generalized Voronoi Diagram (GVD) using the GPU framebuffer. (b) Close-up. Instead of copying the entire framebuffer to the CPU, we copy only the grid points that are relevant to the GVD (white disks). (c) By subdividing large buffers into multiple tiles, we lift the algorithm to virtually infinite resolutions. (d) The ECM navigation mesh is obtained by marking* event points *(small dots) where obstacle normals are intersected, and adding closest obstacle points (gray segments) to event points and edge endpoints. We compute most of this data on the GPU.*

Abstract

This paper presents a GPU-accelerated approach for improving the approximated construction of Generalized Voronoi Diagrams (GVDs). Previous work has shown how to render a GVD onto the GPU framebuffer, and copy it to the CPU for extraction of a high-quality diagram. We improve upon this technique by performing more computations in parallel on the GPU, and reducing the amount of data transferred to the CPU. We also design a multi-tiled construction technique that overcomes hardware limitations and enables much higher rendering resolutions, thus reducing discretization errors. Next, we extend our approach to create an Explicit Corridor Map navigation mesh, which is an efficient data structure for path planning in modern crowd simulation systems. The new implementation allows much faster construction of GVDs and navigation meshes at virtually infinite resolutions.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors

Keywords: Voronoi diagram, navigation mesh, GPGPU

*e-mail:rudi@textkernel.nl
†e-mail:W.G.vanToll@uu.nl
‡e-mail:R.J.Geraerts@uu.nl

Motion in Games 2014, November 06 – 08, 2014, Los Angeles, California.

ACM 978-1-4503-2623-0/14/11 $15.00
http://dx.doi.org/10.1145/2668084.2668093

1 Introduction

An important goal of modern interactive virtual environments is to efficiently and naturally steer many agents. To compute realistic paths for these agents, it is common to run search algorithms on a compact and efficient representation of the walkable space, referred to as a *navigation mesh*.

One particular navigation mesh is the Explicit Corridor Map (ECM) [Geraerts 2010], which is related to the Generalized Voronoi Diagram (GVD) of the environment's obstacles. ECMs allow efficient computation of paths with any desired clearance from obstacles, and are thus capable of steering thousands of heterogeneous characters through multi-layered 3D environments in real-time, with many possible extensions such as crowd density constraints, weighted regions, and dynamic updates [Jaklin et al. 2013]. The method is used in various crowd simulation packages and research projects.

ECMs can be constructed using the GPU framebuffer, based on an approximated computation of the GVD [Hoff et al. 1999]. The results of this algorithm depend on the *resolution* of the framebuffer: in general, a higher resolution improves the quality of the mesh, but leads to longer construction times. Furthermore, this resolution has an upper limit imposed by GPU and CPU memory constraints.

In this paper, we describe a new, more efficient approach to compute GVDs and ECMs at very high, virtually infinite resolutions. This approach takes advantage of general-purpose GPU (GPGPU) computing to improve performance and overcome hardware limitations. Our main *contributions* are the following:

- We show how to compute approximated GVDs more efficiently by employing GPGPU techniques. This dramatically decreases running times and memory usage.
- We describe a multi-tiled technique that lifts this algorithm to virtually infinite resolutions, thus reducing precision errors.
- We extend the algorithm to compute high-quality navigation meshes.

2 Preliminaries and Related Work

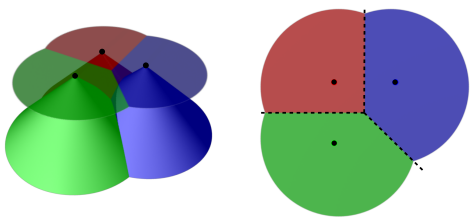
2.1 Navigation Meshes

In games and simulations, a common task is to let AI-controlled characters compute paths through the virtual environment. To allow efficient and flexible path planning, one can use a *navigation mesh*: a subdivision of the environment's walkable space into polygonal regions. When a character plans a path through the mesh, it can use the surrounding free space to locally adjust its movement during the simulation. Navigation meshes used to be constructed 'by hand', but automatically computed navigation meshes are surging, both for 2D environments [Wein et al. 2005; Geraerts 2010; Kallmann 2010] and for multi-layered 3D environments, in which multiple 2D layers are connected [Pettré et al. 2005; van Toll et al. 2011; Oliva and Pelechano 2013; Mononen 2014].

2.2 Voronoi Diagrams and Approximations

A fundamental data structure in computational geometry is the 2D *Generalized Voronoi Diagram* (GVD). Given a planar environment with 2D objects, the GVD is a subdivision of the environment into object-free regions such that each region has a distinct closest object. In this paper, we assume that all objects (or *obstacles* in the context of navigation meshes) are points, line segments, or convex polygons. Non-convex polygons can easily be decomposed into convex parts during pre-processing [de Berg et al. 2008]. Computing an *exact* GVD involves many geometric tests that are numerically unstable [Liotta et al. 1998]. Software packages such as Vroni [Held 2011] work around these difficulties, but they acknowledge the practical complexity of GVDs.

An alternative way to obtain a GVD is to use graphics hardware. Hoff et al. [Hoff et al. 1999] developed a technique that defines a 3D *distance mesh* of cones and quads for each input object, and then rasterizes all distance meshes in distinct colors. An orthographic top view of these objects yields a visual representation of the GVD, as shown in Figure 2. The resulting color information can be converted to a graph structure by copying the framebuffer data to the CPU, and then grouping the relevant pixels into vertices and edges [Geraerts and Overmars 2008]. Multiple improvements and alternatives for the approach by Hoff et al. have been presented [Sud et al. 2005; Fischer and Gotsman 2006].



(a) *Distance meshes* **(b)** *2D top view*

Figure 2: *The Voronoi diagram of a set of objects can be approximated by (a) defining 3D distance meshes per object, and then (b) rendering them from an orthographic top view.*

This approximating algorithm is robust and efficient, but it depends on a *resolution* parameter in multiple ways. First, approximation errors can occur, since objects, vertices, and edge points are rounded to pixel coordinates [Denny 2003]. Second, increasing the resolution will also increase the construction time, because more memory needs to be allocated on the CPU and transferred from the GPU, and more edge pixels need to be traced when converting the buffer to a graph. Third, the resolution of the GVD is limited by the maximum size of a continuous buffer that can be allocated on the CPU. In this paper, we describe how to overcome many of these problems.

2.3 Application: Explicit Corridor Map

The Explicit Corridor Map (ECM) is a navigation mesh for path planning and crowd simulation [Geraerts 2010]. It is based on the *medial axis*, which is the set of all points with at least two distinct closest obstacle points in the input environment [Preparata 1977]. The points of the medial axis form a graph: *vertices* occur in concave corners (vertex degree 1) and at positions with more than two distinct closest obstacle points (vertex degree > 2), while the other points form the *edges* connecting those vertices. The medial axis is a pruned version of the cell boundaries induced by a GVD, as shown in Figure 1d.

The medial axis can be converted to an ECM by annotating specific *event points* with their closest obstacle points. This annotated graph induces a subdivision of the walkable space into polygonal areas, and hence, a navigation mesh. Geraerts [2010] has shown that this mesh is compact, geometrically complete, and efficient to construct and query; furthermore, it can compute paths with arbitrary clearance from obstacles. The ECM has also been extended to multi-layered 3D environments [van Toll et al. 2011]. Incremental Voronoi diagram algorithms can be used to update the ECM dynamically in real-time [van Toll et al. 2012].

Figure 1d shows an example of an ECM in 2D. The ECM is computed using the graphics-based GVD technique described earlier, followed by post-processing steps for pruning non-medial axis edges and adding closest-obstacle annotations [Geraerts 2010]. We show how GPGPU techniques can be used to improve construction of both GVDs and ECMs.

2.4 GVDs and Navigation Meshes on the GPU

It would be interesting to compute GVDs and navigation meshes directly on the GPU, with no further processing of data on the CPU. Rong and Tan [2006] have used jump flooding to compute Voronoi diagrams on the GPU. NVIDIA developed a GPGPU-powered navigation mesh for its AI framework [Bleiweiss 2010], but it does not have the same flexibility as the ECM (e.g. it is less compact and it does not support characters of arbitrary radii). The same paper states that graph operations similar to our tracing step are hard to parallelize effectively on the GPU. For this reason, we keep using the CPU for converting pixels to compact ECM edges. In this paper, we focus on *pre-processing* the visual GVD approximation on the GPU as much as possible, such that less data needs to be copied to and processed on the CPU.

More recently, Hong et al. [2011] have developed an advanced graph search technique that subdivides CUDA workloads into warp units, allowing to parallellize clearly divergent parts over group of threads. This may allow us to perform efficient tracing on the GPU; we will investigate it in the near future.

3 GPGPU-Enhanced GVD Construction

This section shows how GPGPU techniques can pre-process the GVD approximation stored in the framebuffer, to reduce the CPU load afterwards. We assume that the GPU stores the framebuffer data and the obstacle coordinates grouped into segments (i.e. pairs of two points, plus a reference to the original obstacle). We define a grid of $w_g \times h_g$ points over the pixels, with the coordinate axes starting in the top-left corner. Each grid point represents a point shared by the edges of 4 surrounding pixels. To each grid point (x, y), we can associate a unique ID computed as $y \cdot w_g + x$.

Hence, the entire framebuffer can be seen as an array of n grid points: $FB = \{p_0, p_1, \ldots, p_{n-1}\}$. We will use the terms 'grid point' and 'pixel' interchangeably throughout this paper, because they are so tightly related.

We use GPGPU for the following steps:

- (Section 3.1) Discriminating which pixels are potential vertices or edge points of the GVD;
- (Section 3.2) Filtering out irrelevant pixels that do not need to be copied to the CPU;
- (Section 3.3, ECMs only) Computing closest obstacle points to pixels;
- (Section 3.4, ECMs only) Computing information related to obtacle normals.

Other tasks are still performed on the CPU, such as converting sampled edges to lists of ECM event points, and filtering out GVD edges that do not belong to the medial axis. These steps are described in previous work [Geraerts 2010]. We reduce the CPU workload for these steps by precomputing the required information on the GPU (Sections 3.3 and 3.4).

3.1 Marking Vertices and Edge Points

Each grid point has four neighboring pixel colors (except near the image borders). By checking the number of different colors we can derive the point's role in the GVD: 2 colors signal a point that can be part of a GVD edge while 3 or 4 colors signal a potential vertex. We will call the number of distinct neighboring colors the 'degree' of a point. Note that this is not necessarily the same as the degree of the corresponding GVD point: for instance, edge endpoints that lie on a concave obstacle corner will become a degree-1 GVD vertex, but they are represented by degree-2 pixels.

After rasterizing the distance meshes, we keep the data on the GPU and invoke a CUDA kernel that computes the degree for each point. We also keep two *marker buffers* that store a 1 at all pixels marked as a vertex or an edge point, respectively. We subdivide the framebuffer into 2D CUDA blocks, eventually spawning one CUDA thread per pixel for maximum parallellism. The degree of a pixel could also be estimated by activating and processing a z-buffer [Geraerts 2010], but since this would involve consuming additional GPU memory and/or performing additional memory transfers, it would hinder performance in our GPGPU pipeline.

3.2 Filtering Pixels

As can be seen from Figures 1a and 1b, many pixels in the framebuffer have degree 1 and are unnecessary for obtaining the underlying graph of the GVD. Previously, the entire framebuffer was transferred to the CPU [Geraerts 2010]. To save time and memory, we filter out the irrelevant pixels by compressing the two marker buffers into smaller arrays containing only the relevant pixels (marked with a 1).

To perform this compression efficiently on the GPU, we compute the *exclusive prefix sum* of both marker buffers. The exclusive prefix sum of a sequence $S = \{x_0, x_1, \ldots, x_{n-1}\}$ is defined as $P = \{0, x_0, x_0 + x_1, \ldots, \sum_{i=0}^{n-2} x_i\}$. It can be computed efficiently in parallel on the GPU [Merrill and Grimshaw 2011]. In our case, if S is a marker buffer, and $S[i]$ is marked with a 1 (i.e. the ith pixel is a potential GVD vertex or edge point), then $P[i]$ contains the desired index of the ith pixel in the compressed array. Furthermore, the total number of marked pixels can easily be computed as $P[n-1] + S[n-1]$ (the last element of the exclusive prefix sum, plus a possible 1 for the last pixel). With this information, we can efficiently create two arrays of the correct size containing only the marked pixels. A summary is shown in Figure 3.

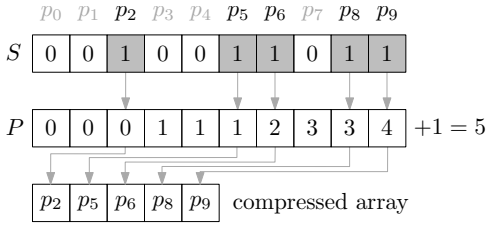


Figure 3: *From an array S of pixels marked with 0 or 1, we compute the exclusive prefix sum P. For all pixels p_i marked with 1, $P[i]$ stores how many 1s have occured so far. This allows us to build a compressed array containing only the 1-pixels.*

3.3 Computing Closest Obstacle Points

The previous step results in two arrays of pixel IDs; from now on, we refer to these as the 'marked pixels'. They already provide enough information to obtain a sampled graph representation of the GVD [Geraerts and Overmars 2008]. From this point on, we discuss extra steps that lead to an Explicit Corridor Map, whose underlying graph is a compact representation of the medial axis. An explanation of how to filter non-medial-axis edges from the GVD can be found in previous work [Geraerts 2010].

Recall from Section 2.3 that the ECM should store the nearest obstacle coordinates at selected *event points*. In practice, it is useful to store the closest obstacle points for all marked pixels, and then decide which pixels are event points [Geraerts 2010].

Finding the nearest point on a polygon from a query point is normally a non-uniform computation that can perform badly on the GPU. Given any marked pixel, we can check what are the closest input polygons (at most 4) by reading from the associated color information. We use the GPU to quickly compute the closest points of *all marked pixels* to *all segments* of the closest obstacles, and for each polygon we keep the overall closest point to the pixel. Selecting this closest point from a set of candidates can efficiently be parallelized on the GPU; this is commonly referred to as *segmented reduction*. We invoke the kernel spawning enough threads to allow computation of a point-to-segment distance per thread. To prevent threads from having to read the same data simultaneously, we duplicate the obstacle segments in GPU memory m times, where m is the number of marked pixels. While this copying may seem rigorous, preliminary experiments showed that it leads to increased performance, most likely because using different memory locations per thread triggers common GPGPU parallelizations.

3.4 Assigning obstacles' normals for edge pixels

ECM event points occur exactly at the intersections of obstacle normals with the medial axis, as shown in Figure 1d. To find these intersections, we use the GPU to find out between which obstacle normals each edge pixel lies.

First, we compute the outward normals (directional 2D vectors) of all obstacle segments, by spawning a GPU thread for each normal to be computed (Figure 4). Next, we look at all edge pixels. Per pixel, we determine the local edge direction to find out which two colors (i.e. obstacles) lie on the left and right side of the edge. For both resulting obstacles, we determine between which normals the pixel is located, using one GPU thread per query (i.e. two threads

per edge pixel). We assign an index i to the pixel if it lies in the space spanned by obstacle point p_i and normals n_{i-1} and n_i. Otherwise, we assign the index -1. See Figure 4 for an overview. Later, the CPU will traverse the edge pixels and create event points at all pixels where this index changes.

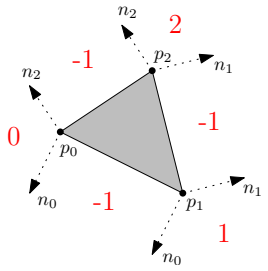


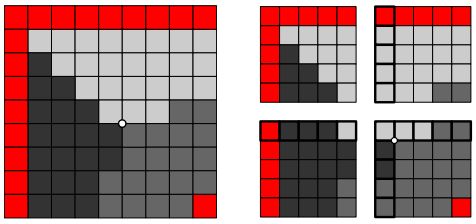
Figure 4: *Normal indexing. For each obstacle (gray), we compute the normals n_i (dotted half-lines). For each edge pixel, we assign an index to indicate between which normals it lies. Event points will be created at pixels where this index changes.*

In this step, we do not copy the obstacle normals in GPU memory: preliminary experiments revealed that it did not improve the performance. Because there are fewer conflicts than in the previous step, the overhead introduced by this 'unrolling' is too high in this case.

4 Multi-Tiled GVD Extension

The method described up until now is limited by a maximum resolution for the GPU framebuffer. To overcome this, we have developed an extension that separates the grid into multiple tiles that do fit in the framebuffer. Since we compute a GVD or ECM per tile and then stitch the graphs together without further need of the framebuffer, we are no longer limited by the framebuffer size; this allows much higher resolutions. We use the same level of precision throughout the entire environment.

The main difficulty of this extension lies in connecting the adjacent tiles. First, we ensure that there is a 1-pixel overlap between the tiles, such that the degree of each pixel can be computed properly: see Figure 5. To prevent duplicate work for the overlapping pixels, we do not process the pixels of the first row and column of each tile. In our application, it is not a problem to ignore the first row and column of the entire grid: since all environments have a bounding box, the global border always consists of obstacle pixels, so it can never contain GVD elements.



(a) *Color data (close-up)* **(b)** *Multiple tiles*

Figure 5: *Multi-tiled GVDs. (a) Top-left corner of the color data. The first row, first column and bottom-right pixel are obstacles. Voronoi regions are shown in different shades. A Voronoi vertex occurs near the middle of the image. (b) A subdivision into tiles. Overlapping pixels are highlighted with thick edges. Without a 1-pixel overlap between tiles, the vertex would not be recognized.*

Second, we need to merge the GVD edges around tile borders. We do this entirely on the CPU as a final step because it involves case distinction and object grouping that is hard to parallellize. After we have performed all the steps from Section 3 for each tile, we have (on the CPU) a set of partial GVDs. However, most edges that end at a tile border are likely to be incomplete, because they continue in the adjacent edge. For each edge endpoint e_1 at the border of a tile, we check the neighbouring grid point e_2 from the adjacent tile. The following cases can occur:

- e_1 and e_2 are both edge points. The two partial edges are merged into one longer edge.
- e_1 is an edge point and e_2 is a vertex (or vice versa). The edge is extended to e_2 such that it ends at the vertex.
- e_1 and e_2 are both vertices. An extra edge (with a length of 1 pixel) is added between the two vertices.

When we have processed all edges in this fashion, the result is a single, high-resolution GVD for the entire environment.

5 Implementation Details

We have implemented the GPGPU algorithm in C++ using the CUDA SDK and version 1.5.0 of the Thrust library [Hoberock and Bell 2010]. Our software extends an ECM framework that already provided a solid implementation of graphics-based ECM construction [Geraerts 2010]. Since we use Thrust mainly for routines for which GPGPU-optimized approaches are well-known (e.g. prefix sums and parallel filtering), our improvements could also be implemented using other platforms such as OpenCL.

The implementation tries to take advantage of GPGPU in many ways. First, when computing the degree of a grid point, we use texture memory to speed-up the reads of neighboring pixels, since texture queries are cached and optimized for 2D spatial locality.

Second, we allow memory coalescence, i.e. having a sequence of threads efficiently access a sequence of memory blocks. To maximize the number of coalesced reads and writes during the various steps, we arrange our data in a 'Structure of Arrays' fashion instead of the more traditional 'Array of Structures'. For example, an Array of Structures for 2D points would contain x and y coordinates interchangeably:

$$P = \{x_1, y_1, x_2, y_2, \ldots, x_n, y_n\}$$

Instead, we let two arrays store the x and y coordinates separately:

$$X = \{x_1, x_2, \ldots, x_n\}, Y = \{y_1, y_2, \ldots, y_n\}$$

We apply this to obstacles, vertex positions, closest points, and obstacle normals. This way, when a GPU thread t_i needs to read a point (x_i, y_i), it only needs to looks at one element of each array. Consequently, more threads can read from the same array in parallel, and the GPU can group these read operations more efficiently. This technique is typical for GPGPU programming.

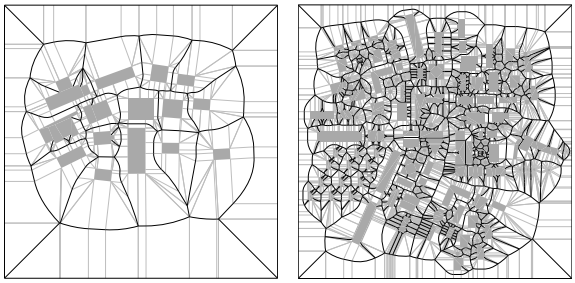
Third, we use 'pinned' (i.e. page-locked) memory on the CPU to mitigate the cost of transfers from GPU to CPU. On modern systems with a front-side bus, the bandwidth between pinned host memory and device memory is higher.

6 Results

We have used the new implementation to construct the GVD and ECM of different environments at various resolutions, and we compared the memory usage and speed of the new approach to an existing one [Geraerts 2010]. Our machine had the following specifications: Intel i7-3930K CPU at 3.2 GHz, NVIDIA GeForce GTX 680

GPU, 16 GB RAM, and Windows 7 64-bit as its OS. The software was developed and built using Visual Studio 2010.

We used the two environments shown in Figure 6. *Military* is a simple environment that measures 200 × 200 m and features 23 convex obstacles. *City* measures 500 × 500 m and features 548 convex obstacles that induce many routes and narrow passages.



(a) *Military* **(b)** *City*

Figure 6: *The environments used in our experiments, along with their medial axis (black curves) and closest-obstacle data (gray segments).*

For convenience, we used multiples of 1024 pixels in width and height at all resolutions. We used the multi-tiled technique purely to enable higher resolutions: to prevent unnecessary graph stitching operations, it is always good to use the largest possible tile size. The maximum tile size on our machine was 4096 × 4096 pixels; hence, we performed multi-tiling for all ECMs at higher resolutions.

6.1 Running Times

Figure 7 summarizes the construction times recorded. All times are averaged over 40 runs. With respect to the existing implementation [Geraerts 2010], the GPGPU version achieved speedups of up to 8.5x for Military and 3.5x for City. Moreover, the introduced merging phase has basically no impact on the total times: it consists for less than 2% of running times also when 64 tiles of 4096 squared pixels are employed (worst case). A two-tailed Student t-test confirmed that the new implementation is significantly faster ($\alpha = 0.05$) at all resolutions from 4096 × 4096 pixels and higher. The speedup factor depends on the complexity of the ECM that is being constructed; City has a more complex ECM, i.e. more pixels to copy and edges to trace. Hence, the new implementation is more 'output-sensitive' and less heavily dependent on the resolution.

6.2 Memory

We have also measured the memory transferred from CPU to the GPU, as shown in Figure 8. We use 32-bit colors when rasterizing the GVD, because it allows us to handle a greater number of obstacles (i.e. 2^{32}). Since the original implementation simply transfers all pixels of the GPU framebuffer, it transfers $w_g \times h_g \times 4$ bytes. This cost does not depend on the input environment, but only on the resolution. For instance, 16 MB was already needed at 2048×2048 pixels, and this amount grows rapidly as the resolution increases.

By contrast, in the GPGPU implementation the amount of transferred memory depends on the complexity of the graph to be traced. The City environment transfers only 1 MB at the same resolution; it transfers 16 MB at a resolution of 16384 × 16384 pixels. Multi-tiling does not affect this significantly since we only transfer an additional strip of pixels for each border shared by two tiles.

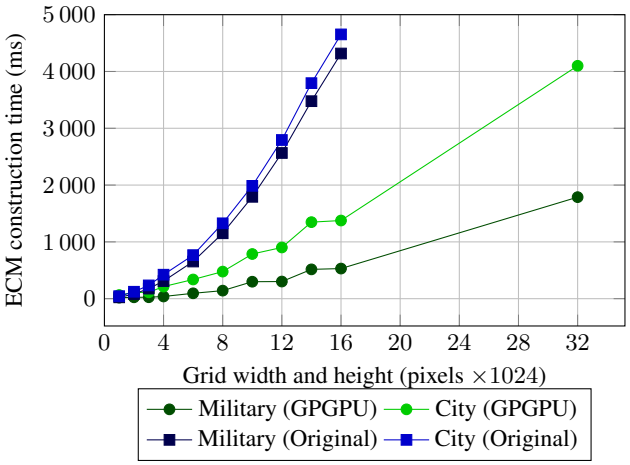


Figure 7: *Running times of the implementations at various resolutions. Our GPGPU implementation exhibits significantly lower computation times at most resolutions. It also allows higher resolutions that were previously impossible due to buffer size constraints.*

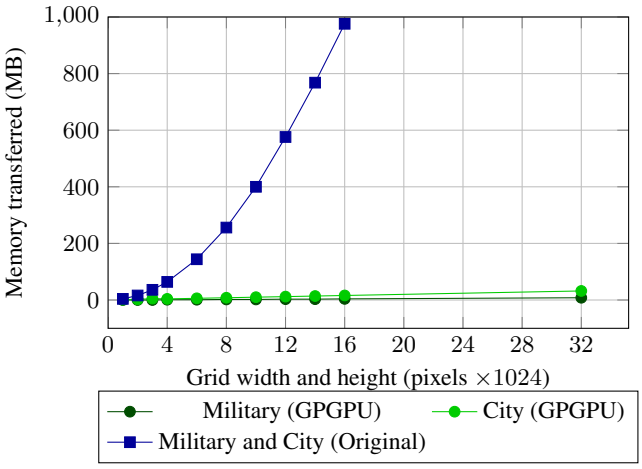


Figure 8: *Memory transferred from the GPU. In the original implementation, the amount of memory depends only on the grid resolution. In comparison, our GPGPU implementation transfers an almost negligible amount of memory.*

6.3 Resolution and Precision

Multi-tiling allows resolutions that were originally not possible. After all, the other implementation copies the entire color buffer to the CPU, which imposes a limit on the maximum buffer size: it cannot produce results at more than 32,000 squared pixels using 32-bit colors. The new method is free from this constraint, so we can now compute GVDs and ECMs that capture the environment's geometry much more accurately.

Mapping the GVD to a discrete grid of insufficient size can lead to mistakes. For example, narrow passages may not be detected, short edges may be overlooked, or normals may be crossed multiple times. The existing ECM implementation can already filter out incorrect configurations: the method is *robust* in the sense that it always generates a navigation mesh that allows artifact-free path planning of agents. However, it cannot fill in information that was lost during the discretization. A higher resolution decreases the number of errors *and* allows small details to be recognized.

7 Conclusions and Future Work

In this work, we have shown how to take advantage of GPGPU methodologies when computing a Generalized Voronoi Diagram (GVD). A previous implementation rendered an approximation of the GVD on the GPU, transferred all data to the CPU, and performed all computations there. Our new implementation assigns more work to the GPU: we use parallelism to mark relevant vertices and edges, to decide which pixels need to be copied to the CPU, and to compute closest obstacle points. This significantly reduces running times and CPU memory usage. Time and memory consumption become less dependent on the resolution, and more related to the complexity of the GVD that is being computed.

We have also introduced an approach that builds the GVD in separate tiles and then stitches them together. This overcomes the technical limitations that imposed an upper bound on the GVD resolution. Since the resolution influences the precision of the computed data structure, our multi-tiled technique can greatly reduce approximation errors. Our results were obtained using only a single 'prosumer' graphics card, a baseline for multimedia-rich applications.

The GVD is easily extended to the Explicit Corridor Map, which is a state-of-the-art navigation mesh for 2D and multi-layered 3D environments. Hence, our implementation provides a step forward in building a framework for interactive applications that uses graphics hardware not only for rendering or physics, but also for AI and geometrical computations. At the same time, the decreased CPU load makes this approach more applicable for memory-intensive applications such as level editors or modeling tools.

The improved efficiency could be investigated further with the goal of computing ECMs at interactive rates. Technically, we could try to perform memory transfers and GPGPU computation in parallel, evaluate the use of GPU persistent threads, or address tracing on the GPU using warp-based methods. We are also interested in designing a GPGPU-friendly approach for *partial* re-computations of the changes in real-time. This would allow us to process dynamic obstacles in a similar manner. In general, we expect that this work will encourage researchers to further explore GPGPU approaches for navigation meshes and other AI purposes.

References

DE BERG, M., CHEONG, O., VAN KREVELD, M., AND OVERMARS, M. 2008. *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer-Verlag.

BLEIWEISS, A. 2010. Parallel compact roadmap construction of 3D virtual environments on the GPU. In *Intelligent Robots and Systems, 2010 IEEE/RSJ International Conference on*, IEEE, 5007–5013.

DENNY, M. 2003. Solving geometric optimization problems using graphics hardware. In *Computer Graphics Forum*, vol. 22, Wiley Online Library, 441–451.

FISCHER, I., AND GOTSMAN, C. 2006. Fast approximation of high-order Voronoi diagrams and distance transforms on the GPU. *Journal of Graphics Tools 11*, 4, 39–60.

GERAERTS, R., AND OVERMARS, M. 2008. Enhancing corridor maps for real-time path planning in virtual environments. In *International Conference on Computer Animation and Social Agents*, 64–71.

GERAERTS, R. 2010. Planning short paths with clearance using Explicit Corridors. In *IEEE International Conference on Robotics and Automation*, 1997–2004.

HELD, M. 2011. Vroni and ArcVroni: Software for and applications of Voronoi diagrams in science and engineering. In *International Symposium on Voronoi Diagrams in Science and Engineering*, 3–12.

HOBEROCK, J., AND BELL, N., 2010. Thrust: A parallel template library. Version 1.5.0.

HOFF, III, K., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. *International Conference on Computer Graphics and Interactive Techniques*, 277–286.

HONG, S., KYUN KIM, S., OGUNTEBI, T., AND OLUKOTUN, K. 2011. Accelerating CUDA graph algorithms at maximum warp. *SIGPLAN Notices 46*, 8, 267.

JAKLIN, N., VAN TOLL, W., AND GERAERTS, R. 2013. Way to go - A framework for multi-level planning in games. In *Planning in Games Workshop*, 11.

KALLMANN, M. 2010. Navigation queries from triangular meshes. In *Proceedings of the 3rd International Conference on Motion in Games*, 230–241.

LIOTTA, G., PREPARATA, F., AND TAMASSIA, R. 1998. Robust proximity queries: An illustration of degree-driven algorithm design. *SIAM Journal on Computing 28*, 3, 864–889.

MERRILL, D., AND GRIMSHAW, A. 2011. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters 21*, 02, 245–272.

MONONEN, M., 2014. Recast Navigation. `https://github.com/memononen/recastnavigation`.

OLIVA, R., AND PELECHANO, N. 2013. NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments. *Computers & Graphics 37*, 5, 403–412.

PETTRÉ, J., LAUMOND, J., AND THALMANN, D. 2005. A navigation graph for real-time crowd animation on multilayered and uneven terrain. In *Proceedings of the First International Workshop on Crowd Simulation*, 81–89.

PREPARATA, F. 1977. The medial axis of a simple polygon. In *Mathematical Foundations of Computer Science*, vol. 53. Springer, 443–450.

RONG, G., AND TAN, T. 2006. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *Symposium on Interactive 3D graphics and games*, ACM, 109–116.

SUD, A., GOVINDARAJU, N., AND MANOCHA, D. 2005. Interactive computation of discrete generalized Voronoi diagrams using range culling. In *International Symposium on Voronoi Diagrams in Science and Engineering*.

VAN TOLL, W., COOK IV, A., AND GERAERTS, R. 2011. Navigation meshes for realistic multi-layered environments. In *IEE/RSJ International Conference on Intelligent Robots and Systems*, 3526–3532.

VAN TOLL, W., COOK IV, A., AND GERAERTS, R. 2012. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds 23*, 6, 535–546.

WEIN, R., VAN DEN BERG, J., AND HALPERIN, D. 2005. The Visibility-Voronoi complex and its applications. In *Proceedings of the 21st Annual ACM Symposium on Computational Geometry*, 63–72.