Geodesic Distance Computation via Virtual Source Propagation

supplemental material

1 Basic Implementation

We added a basic implementation of our method as a standalone C++ source file (`main.cc`). A minimal `CMakeLists.txt` is also provided to ease the building process. Building requires a C++17 compiler, we personally tested it on clang 7.

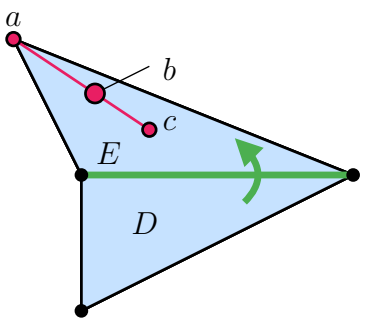
The program takes a path to an obj mesh and the index of the source vertex. The mesh is loaded, geodesics (with simple timings and stats) are computed for the source vertex. The result is printed to stdout. We also added a few example meshes.

We intend to publish a complete C++ library featuring many of the described extensions and applications.

2 Data-Driven Visibility Heuristic

In Section 3.2 of the paper, we described a simple heuristic for computing which reference point is used for determining visibility. A few more details on how the parameters are optimized follow.

To recap: Let c be the center of gravity of triangle E and a the opposite corner to triangle D. Then we check virtual source visibility for a point $b = \lambda \cdot c + (1 - \lambda) \cdot a$ between c and a. The point b acts as a kind of visibility probe. The blending weight λ is derived as a function of some anisotropy measures on E.



More concretely, we compute the following four anisotropy measures for the triangle E given its edge lengths e_i (shared edge is e_1) and height h of triangle E on e_1:

$$\tau_1 = \frac{\max e_i}{\min e_i}, \quad \tau_2 = \frac{\max e_i}{e_1}, \quad \tau_3 = \frac{h}{\max e_i}, \quad \tau_4 = \frac{h}{e_1}.$$

By comparing them to four thresholds $\tau_1^* \dots \tau_4^*$, we obtain a four-bit binary signature (i.e. 16 possible patterns). Via a small look-up table we get the blending weight λ. This procedure has 20 parameters (4 thresholds plus 16 look-up table entries).

We use a simple genetic algorithm to optimize these parameters once and then keep them fixed for all meshes. The algorithm is a $(1 + 1)$-ES (evolution strategy) and works as follows: Given the current 20 dimensional parameter vector, we add random noise to each entry. Then, the fitness is evaluated of the "mutated" parameter vector. If this fitness is better than the previous one, the mutant replaces the original, otherwise the mutant is discarded. This is of course extremely primitive (and basically a stochastic search). Nevertheless, it can effectively find good minima in our search space, especially as there are no gradients for this heuristic (it only influences a binary decision).

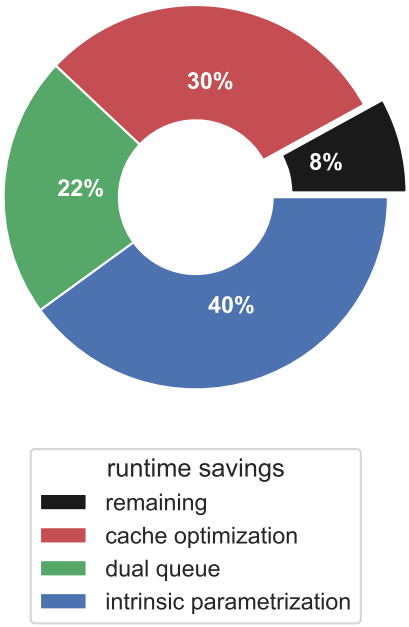
The fitness function is also kept relatively simple: We take 20 random, representative meshes from the Thingi10k and the Tet10k dataset. Our method is then evaluated with the current parameter vector and the average geodesic error is computed for 20 random sources (averaged over all sources, vertices, and meshes). The negative error is our fitness (higher is better).

A few experiments suggested that taking more meshes for the fitness only slows down the search but does not improve the quality gain noticeably. We theorize that further gains can be achieved by changing the structure of the heuristic or the potential points it computes (not restricted to the line between a and c). We left this for future work. Overall, this heuristic generalized reasonably well and lead to approximately 20% less average error in our experiments.

Our pre-optimized parameters can be found in line 411–416 of `main.cc`

3 Performance Breakdown

The inset shows the relative effects of our optimizations. Our final algorithm takes about 400–500 cycles per vertex. Note that we propagate over halfedges and only continue if a shorter path was found. This translates to roughly 4.6 update steps per vertex.

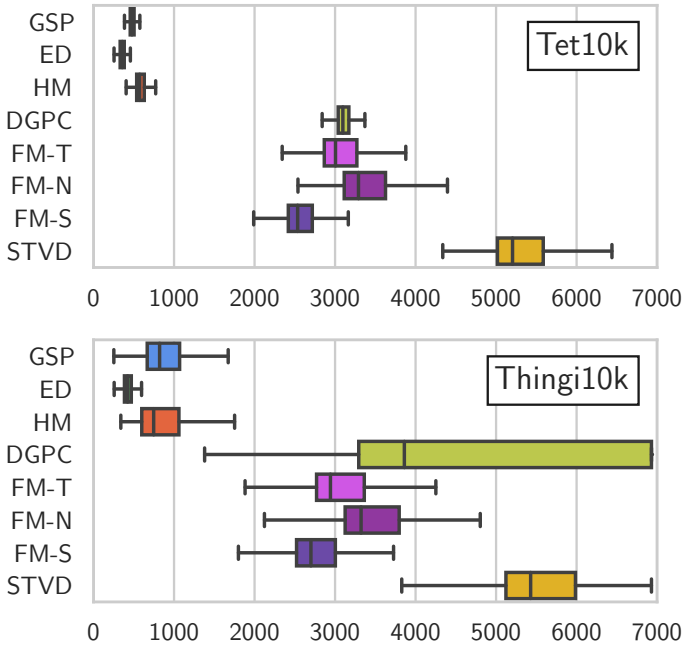


The dual queue approach is applicable for any algorithm with a cheap update step as the priority queue might dominate otherwise. In our case we save around 1300 cycles per update step. For example, Table 1 in the paper shows that Dijkstra would significantly benefit from the dual queue as well. The inset in Section 4.2 in the paper compares different queue strategies. Table 2 in the paper highlights the importance of using the dual queue approach with GSP: using a priority queue is 4× slower.

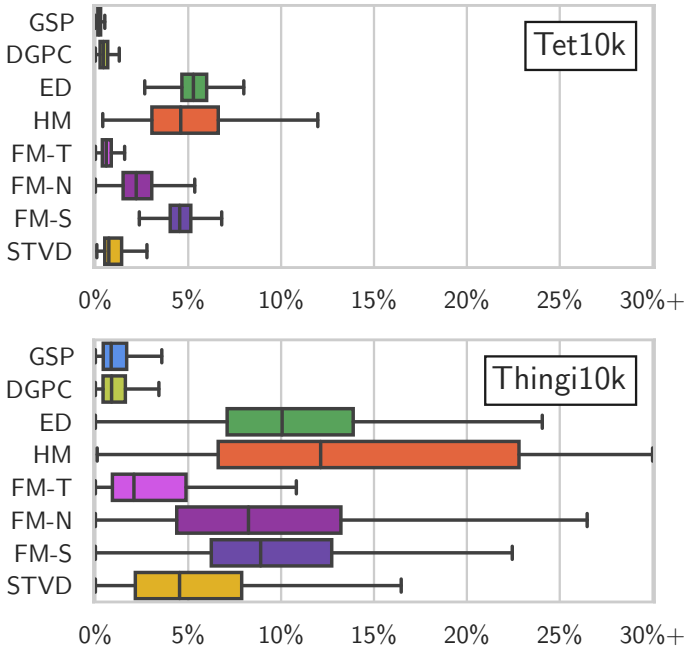
In Section 4.1 of the paper, we argued that the intrinsic parametrization is more efficient, as an explicit implementation requires edge unfolding. Without the intrinsic parametrization, a basis change matrix must be applied and some simplifications of the intrinsic version do not apply. When counted per vertex, this costs an additional 2000–4000 cycles depending on the implementation. Note that while this particular intrinsic formulation is specific to our approach, window propagation methods use a similar formulation.

Memory layout affects every method as can be seen in Table 1 in the paper. For GSP, not optimizing the memory layout costs 800-3000 cycles per vertex more. Figure 6 in the paper illustrates this effect where the algorithm is over 5× more expensive in the worst case. While designed to accelerate propagation-based algorithms, PDE-based methods also benefit because cache optimizations tend to correlate with fill-in reduction. For a fair comparison, all algorithms were tested on layout-optimized meshes in the evaluation.

Mesh anisotropy has a noticeable negative effect on approximation error and speed, which is shown in Figure **??**. For best results, triangles should be not exceed $1:2$ ratio between shortest and longest edge per triangle.



(a) Performance measured in CPU cycles on a single 4.5 GHz core for single-source-all-targets, divided by number of mesh vertices.



(b) Accuracy measured as relative error ($|d_{\text{computed}} - d_{\text{real}}| / d_{\text{real}}$), averaged over all mesh vertices.

Figure 1: Performance and accuracy evaluated over the benign `Tet10k` data set (about 6400 meshes) and the malign `Thingi10k` data set (about 3400 meshes). For each mesh, the same 20 experiments with a single source located at a random vertex were performed. The data is visualized in box plots showing three quartiles and whiskers at the standard 1.5 interquartile range. The compared algorithms are our GSP, Dijkstra on edge lengths (ED), the heat method (HM), fast marching from Sethian (FM-S), Novotni (FM-N), Tang (FM-T), discrete geodesic polar coordinates (DGPC), and the short-term vector Dijkstra (STVD).

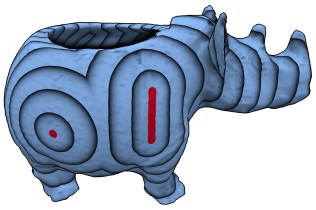
4 Extensions

In Section 4.4 of the paper, we briefly showed several extensions to the base algorithm. Here, these are described in more detail:

4.1 Source Types

The original formulation in the paper Section 3.1 and Section 4.1 treat the virtual geodesic source S as a point. Intrinsically, a point is represented by distances to vertices, $d_{v,T}$.

To support a new type of source, such as lines or circles, two things must be changed: First, the representation must be augmented to support encoding of the desired types. Second, the propagation scheme in Section 4.1 inserts a new virtual geodesic point source if $\overline{CS}$ and $\overline{AB}$ do not intersect. According to Section 3.2, this can be interpreted as a visibility test: Given a reference point Q inside the triangle, we check if the virtual source is "visible" through the edge we propagate over. This interpretation can be easily extended for other primitive types.



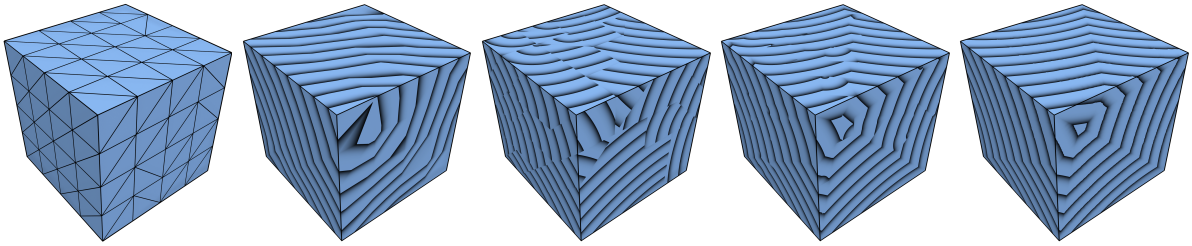
point and line segment source

Line segments are characterized by two points and thus two distances to each triangle vertex. For visibility, segments can be fully or partially visible or hidden. In the first case, the virtual source is not changed. In the second, it is clipped to a smaller segment (the visible part). In the last case, we insert a new virtual point sources.

Similarly, 2D triangles can be used as sources, characterized by their three vertices, and circles, characterized by point and radius. The visibility is slightly more complex in these cases. A simplified handling can be employed at the cost of approximation error.

4.2 Nearest-Neighbor Information

In the multi-source-all-destination scenario it might be important to know which source is the closest for a given vertex. With fast marching or the heat method it is non-trivial to propagate or recover this information. Tracing geodesic paths to their seed by following the shortest distance is not always possible due to local minima produced by such methods. The information cannot be propagated for fast marching either because it is unclear which source should be propagated when the two vertices belong to different sources. In fact, this is exactly the situation where our virtual source propagation excels: GSP always propagates a valid path to a source. Just storing the index of the associated source per face and copying it during the update step is sufficient. For example, we use this extensively during the computation of a centroidal Voronoi tessellation.



(a) (b) (c) (d) (e)

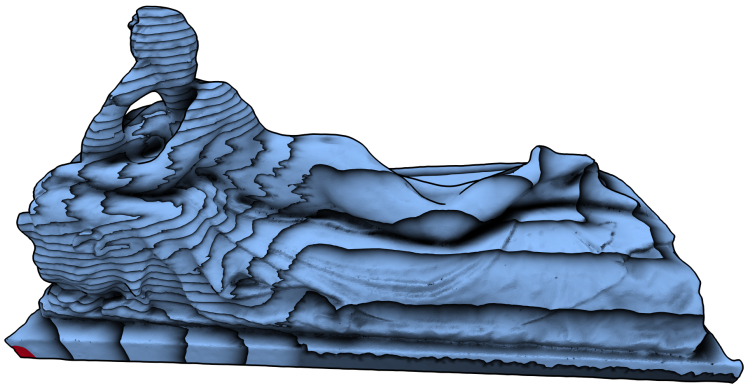
Figure 2: Geodesic distances for a tessellated cube. The source is on the back side. (b) shows linear interpolation of vertex distances. In (c), the per-triangle virtual source is used to compute the distances. The transition between different virtual sources can be refined by also considering neighboring triangles (d). Exact result is shown in (e).

4.3 Reconstruction Accuracy

Given the virtual source information per face, geodesic distances can be reconstructed on the entire mesh with considerably higher accuracy than plain linear interpolation of vertex distances. When given an arbitrary point on the mesh, using the virtual source of the containing triangle is already more accurate than distance interpolation. However, this may produce discontinuities between triangles that have different virtual sources. A better scheme is to use not only the containing triangle but rather the minimum over the three neighboring ones after applying the unfolding. There are cases where reconstruction quality can be improved by taking an even larger neighborhood into account, e.g. the one-rings of all adjacent vertices, but this of course increases the computational cost. Figure 2 shows how the interpolation quality is improved, especially where the influence regions of different virtual sources meet.

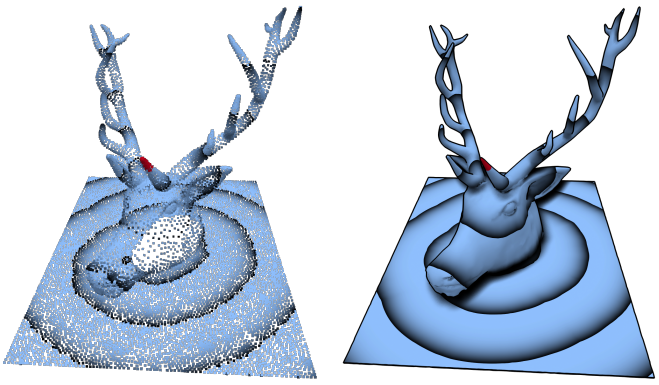
4.4 Anisotropic Metrics

GSP can be trivially adapted to anisotropic domains: Instead of using the original edge lengths, compute the length of each edge in the desired anisotropic metric and use those lengths instead. While a feasible metric should respect the triangle inequality, it can be violated in practice. To make the algorithm stable, all square roots that could be negative, such as $\sqrt{e_3^2 - P_x^2}$, are clamped to zero. The inset shows anisotropic geodesics originating from the bottom-left corner (red). In this example, the right half of the statue's base has only 1% cost. In the upper parts of the left half it is ten times more expensive to move vertically.



region-dependent anisotropic metric

4.5 Unstructured Domains



point cloud mesh reference

Our approach also works on unstructured domains such as point clouds. Instead of enqueuing halfedges we enqueue pairs of vertices. When such a pair is expanded, all vertices in a certain radius or the k nearest neighbors are enumerated. Each enumerated vertex forms a virtual triangle with the pair and performs the regular update step. This is different from a k-NN Dijkstra because the virtual triangle mimics a local surface reconstruction. To prevent infinite looping, each vertex stores its lowest known geodesic distance and the update step only enqueues new pairs if the new distance to the enumerated vertex is smaller then the stored one.

Note that the propagation can produce holes or even get stuck if the k nearest neighbor graph is not properly connected. In our tests, $k = 10$ seems to be sufficient for roughly uniformly sampled point clouds.

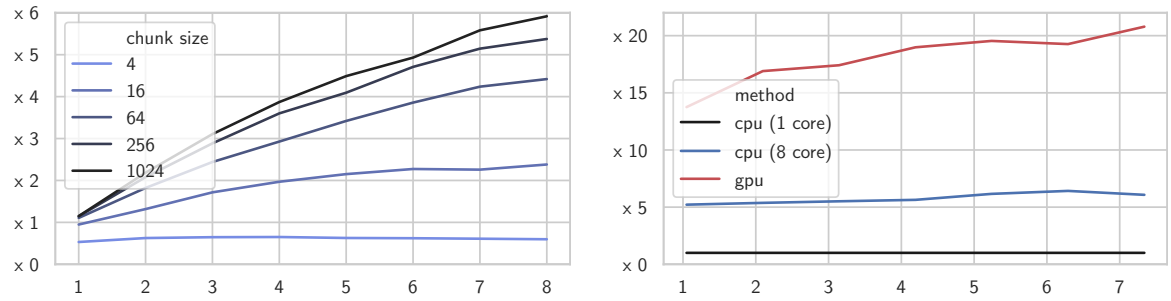
4.6 Higher Dimensions and Mixed Domains

The intrinsic update step described in the paper Section 4.2 can be formulated in arbitrary dimensions. The generic case is an n-dimensional source simplex U and an m-dimensional target simplex T with non-empty intersection $I = U \cap T$. For triangle propagation of meshes in 3D, $n = m = 2$ and I is the shared edge. However, we could also propagate from edges to edges over shared vertices, given a wireframe mesh. In a volume mesh we might want to propagate from a triangle over a shared edge into a cell. In general, not all possible interactions are required. For example, the triangle mesh works fine without considering triangles connected via vertices. Which interactions should be included depends on the use case but as a rule of thumb the "maximal dimension" case should be included such as cell-cell propagation over shared triangles in volume meshes.

The intrinsic parametrization of Section 4.2 operates in two steps: reconstruction in the local coordinate system and intersection test of $\overline{CS}$ with I followed by a projection to I.

For an easy and unambiguous pose of an n-dimensional simplex we can extend the strategy used for triangles: The first point is placed at the origin. The second point is placed on the positive x-axis and is determined by the distance to the first point. The third point is placed on the positive xy-plane and is determined by the distances to the first two points. The k-th point is placed on the positive span of the first $k-1$ axes and is determined by the distances to the first $k-1$ points.

With local coordinates of all points, the intersection and projection can be performed. All involved objects are simplices with simple closed formulas available.



(a) speed-up vs. CPU cores

(b) max speed-up vs. million vertices

Figure 4: For large meshes or many sources, a parallel implementation of our method can lead to significant performance improvements. The first graph plots speed-up against CPU cores on a 4 million vertices mesh with 1000 sources, the second plots the largest observed speed-up against mesh size and compares CPU and GPU implementations.

5 Parallel Implementation

As written in Section 5.8 of the paper, for a parallel implementation, we use the *hybrid postpone* strategy with a *fat queue*. When iterating over queue $\mathcal{A}$, each update enqueues 0–2 new entries in $\mathcal{B}$. The number of new entries correspond to the cases:

- entry is not a shorter path (0 new entries)
- entry was skipped (1 new entry)
- entry is a shorter path and propagates into two neighboring triangles (2 new entries)

In the *fat queue* approach, almost all runtime data is stored in the queue entries. The only changing mesh-stored property is d_T (geodesic distance of triangle center) on which a simple `atomicMin` suffices. An update step can atomically write its new d_T and check if it is a shorter path than previously stored. While multiple entries can try to write to the same d_T at the same time, this is still only local contention.

The only other synchronization point within an iteration is enqueuing new entries to $\mathcal{B}$. Because $\mathcal{B}$ is a simple FIFO queue, enqueuing amounts to incrementing a global atomic counter and using its value as the location in $\mathcal{B}$ to write into. Accessing an atomic is comparatively expensive (50–70 cycles per operation if contended). A better strategy is to process the queue in chunks, for example 64 entries at once. Each update step enqueues into a local queue that is bounded by chunk size $\times 2$. After processing a chunk, a single atomic counter access can be used to allocate space for copying the local queue into B.

On the CPU we use a job-based parallelism model backed by a thread pool. In each iteration, the current queue $\mathcal{A}$ is partitioned into chunks and one job is started per chunk. New entries are first collected into a local queue and then copied to the global queue. Figure 4 (a) shows how chunk size affects the speed-up on a 4 million vertices mesh with 1000 sources. We used an Intel i9-9900K here to show the scaling up to 8

physical cores. Our peak speed-up of factor 6 is only achieved for larger chunk sizes, indicating that the queue contention is indeed a large bottleneck.

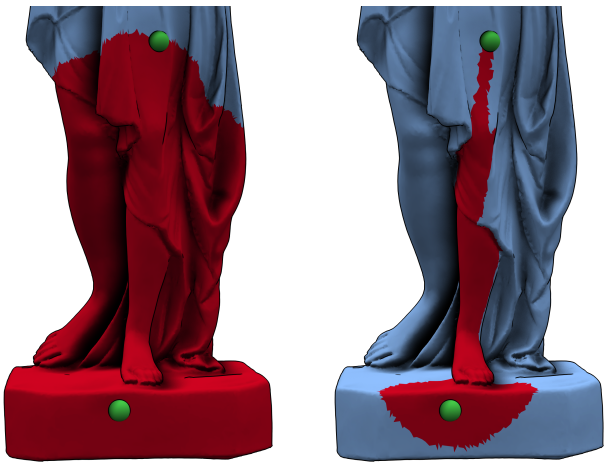
On the GPU we used OpenGL's *Compute Shader*. Apart from minimizing global access, communication with the CPU must be minimized. The graphics card is also more susceptible to dynamic control flow and bad memory access patterns. We store mesh topology, attributes, and the queues in `ShaderStorageBuffer`s. Between iterations, the buffers for $\mathcal{A}$ and $\mathcal{B}$ are swapped. Queue size is also stored in a `ShaderStorageBuffer` and used to start the next iteration via `glDispatchComputeIndirect`. Atomic writes are used to directly update the queue size in this buffer. In this configuration, the CPU is purely sending a command stream to the GPU without needing to wait for results. Every few iterations, the CPU asynchroneously reads the current queue size to know when the algorithm has terminated. *Compute Shader* can also directly use the chunking mechanism via their local workgroup size. The resulting shader works like this:

1. compute propagation info (0–2 new entries), using a workgroup `shared` counter for the total number of entries in the local workgroup
2. wait for other shaders of the same group (`barrier`)
3. if first shader of group, use `atomicAdd` to allocate space for new entries
4. wait for other shaders of the same group (`barrier`)
5. copy entries into output queue (start index is computed in step 3, offset is computed during incrementing the counter in step 1)

As written in the main paper, both parallel implementations have non-negligible overhead, especially compared to our highly optimized single threaded version. The highest speed-up can always be achieved by actually computing multiple problems simultaneously, i.e. if the geodesic computation is used as a building block multiple times. If that is not possible, then our parallel versions can at least provide a speed-up for large meshes or the geodesic fronts are large, e.g. if many start sources were specified. This is for example the case when computing the centroidal Voronoi tessellation on large meshes. Here, we were able to observe a $6\times$ speed-up on the CPU and a $20\times$ speed-up on the GPU.

6 Applications

6.1 Pathfinding



Dijkstra-style
93 000 exp.

A*
7500 exp.

The single-source-all-destination modus operandi is not always desired. Sometimes, a single source-destination path must be found, for example in pathfinding or robotics. Due to the flexibility of label-correcting algorithms, we can tweak or replace the dual queue propagation as we see fit. By terminating the propagation shortly after finding the destination we effectively recover a single-source shortest path algorithm. By replacing the dual queue approach by a priority queue and using the Euclidean distance as heuristic, the classical A* algorithm can be used, cutting down the number of expanded faces drastically. The inset shows the difference between the two variants (expanded faces are highlighted). Dijkstra-style expands about 93 000 faces while A* only 7500.

6.2 Blue Noise / Poisson Disk Sampling

In some applications such as texture synthesis or automatic cage-based rigging, objects should be placed on a surface stochastically but with roughly equal spacing. Such a distribution is sometimes called blue noise and can be generated by Poisson disk sampling or dart throwing.

Our propagation can be stopped at a certain distance by not enqueuing anything above that distance. With this, an efficient algorithm for generating blue noise with a lower-bound radius r can be constructed:

1. Start with zero seeds
2. For each triangle in the mesh (optionally in random order):
   (a) If triangle already assigned, skip it
   (b) Add random point in triangle as seed
   (c) Start propagation from there with maximum distance r
   (while keeping information from all previous runs)

This will progressively fill the mesh with geodesic distance information no larger than r. Because the propagation is stopped early, the total runtime is essentially linear in number

(a) 30 seeds (b) 150 seeds

Figure 6: A geodesic blue noise sampling using stopped-early propagation. The 200 000 faces mesh is shown in the middle and at the end of the sampling procedure. Blue faces are unassigned, shades of red indicate seed assignment. A single-source-all-destination computation takes around 14 ms while the Poisson disk sampling takes 22 ms.

(a) 1 iteration (b) 2 iterations (c) 50 iterations

Figure 7: A centroidal Voronoi tessellation computed with Lloyd's algorithm using our GSP as a sub-component. Different iterations of the CVT with previous seed (green) and updated seeds (red) are shown. For this 280 000 faces mesh, each iteration takes about 50 ms.

of triangles. In our tests, computing such a blue noise sampling is only about twice as expensive as a single-source-all-destination geodesics computation. This distinguishes it from the popular farthest-point-sampling which is vastly more costly. Figure 6 depicts an example of this procedure.

6.3 Geodesic Centroidal Voronoi Tessellation

A centroidal Voronoi tessellation (CVT) is a powerful tool which is well explored in Euclidean space and is of interest in a general manifold setting. The Poisson disk sampling from the previous section provides an excellent starting point for approximating a CVT. A typical approach is to use Lloyd's algorithm. Each relaxation step now consists of:

1. Move each seed point to the center of its Voronoi cell by:
   (a) Using all edges of the cell borders as sources
   (b) Computing geodesics from these edges
   (c) Moving each seed to the farthest point in its Voronoi cell
2. Recompute the Voronoi diagram using GSP

The initialization and each step are about twice as expensive as a single-source-all-destination geodesics evaluation. Note that this algorithm makes heavy use of the nearest-neighbor information as described in Section 4.2. An example is depicted in Figure 7.