Δ—a HPC meshing library (eCSE12-04)

Konstantinos Krestenitis and Tobias Weinzierl
Computer Science, Durham University

Version 0.1, October 8, 2018

Abstract

We present concepts and rationale behind our Δ software, a library to administer and query triangle-based meshes. Δ identifies overlaps, contact points and separation of different meshes. Its simplistic programming interface is supported by algorithms which can exploit modern vector architectures as they combine optimistic iterative geometric algorithms with robust if-based code kernels, which scale on multicore architectures through an intra-library parallelisation plus a strict functional code design, and which support by on-the-fly caching of geometric information. The code has successfully been used in a Discrete Element Method (DEM) and immersed boundary context. Some case studies allow users to quickly start experiments with the code base.

1. Introduction

Handling triangulated meshes is an important task in many scientific simulation codes. To import a mesh—typically from a CAD model or a mesh repository—is one of the first steps in many simulation pipelines, and the coding of appropriate import routines, of mesh mappings onto existing compute grids, of routines that identify where meshes touch is a fundamental technical task. Economic common sense dictates to use a mature software component here. In many traditional setups, the mesh import or generation, respectively, is a sole preprocessing step. Its efficiency thus has not highest priority.

Our work however meanders around two use cases where computational efficiency of the mesh handling is key: Our first example from Discrete Element Methods (DEM). Here, rigid particles as we find them in granulates or powder collide with each other and exchange momentum. We assume that the particles are represented by triangulated meshes [2, 4]. In each and every time step, the meshes thus have to be compared to each other and we have to identify collisions, i.e. contact situations. Non-monolithic fluid-structure interaction with moving meshes can often been read as an generalisation of such a setup. One mesh wraps alike a cushion around the other. It bumps against, i.e. collides with the other mesh. Our second example is dynamically adaptive mesh refinement (AMR) for mesh-based solvers of partial differential equations that are combined with an immersed boundary method [6]. As the simulation mesh or the embedded domain change, the simulation has to identify the distance between newly created degrees of freedom and a boundary mesh on-the-fly to impose the correct boundary terms.

Both demonstrator examples highlight on the functional side the need for mature, easy to use meshing interfaces which allow users to code mesh input and interaction functionality. On the non-functional side, they clarify that it becomes important to deliver geometric information effectively on modern supercomputer architectures. Both demonstrators hereby rely on the same functionality: They handle at least two meshes and they have to know how far these meshes are apart. For an immersed boundary method, such information feeds directly into the computation. For a DEM method, such information identifies contact between structures once we assume that any proximity closer than ϵ denotes contact.

Δ proposes a simplistic programming interface (API) to load meshes, to run primitive mesh modifications such as translations, scaling or rotations, and to compute distances as well as to detect contact and overlap between two meshes. The latter are passed an ϵ parameter, and they return for given input data all contact points, i.e. all points where the input data and the mesh subject of study are closer than ϵ. Input data can be simple points (for the immersed boundary approach, e.g.) or meshes themselves. Our implementation wraps around The Open-Asset Imports-Lib [1] and thus can load various different CAD formats. We focus on triangular meshes here.

Our code is optimised towards modern HPC architectures through various techniques hidden behind the API: (i) We flatten the triangle-edge-vertex graph describing the mesh into a plain series of vertex triples as SoA. The resulting data exhibits some memory overhead as vertices might be replicated, but it can be streamed through the cores without indirect, non-continuous memory access. (ii) We introduce a hybrid collision detection algorithm which combines an iterative Newton-based collision detection with traditional (if-based) distance calculations. The iterative approach vectorises. In cases where it does not converge, we fall back to the robust solve. This results in a vectorised, robust algorithm. (iii) The realisation behind the API runs parallel on multicore systems. It parallelises over triangles. Furthermore, the API itself is thread-safe, i.e. can be invoked from various threads concurrently, and—for point-wise queries as they are used for immersed boundary methods—offers routines that accept whole sets of query points over which it parallelises, too. (iv) Finally, Δ introduces spatial caching: Upon demand or as preprocessing step, we wrap an octree data structure around the mesh data. If geometric proximity data in an ϵ-environment is of interest, distance queries are first compared against the octree. This allows us to identify results further away than ϵ from any geometry quickly. As the octree can be built upon demand, its accuracy and overhead anticipate the structure of the collision queries. For a domain decomposition approach, the cache meta data structure automatically follows the decomposition. To the best of our knowledge, no other geometry library provides this unique combination of efficient realisation ingredients.

The remainder is organised as follows: We first introduce our API (Section 2.) before we discuss details of the used algorithms (Section 3.). Experimental data in Section 4. highlight the potential of our Δ library. We summarise our insight and close the discussion with Section 5.. An appendix (online version on software homepage only) offers how-to information. The software is freely available from [3].

2. Terminology and user programming interface

Δ has a very simple C++ application programming interface (API). At its core is the creation of a Δ mesh instance `delta::Mesh`. Codes hold one mesh instance per mesh administered. While meshes can internally have any representation, they all provide a `flatten` operator which returns the mesh as a sole sequence of triples of vertices.

Example: Flattening (Figure 1) Let mesh A be as graph. Arrays $(x_0, x_1, x_2, \ldots, x_5)$, $(y_0, y_1, y_2, \ldots, y_5)$ and $(z_0, z_1, z_2, \ldots, z_5)$ hold the vertices. A second array of index triples $((0,1,2),(1,2,3),(2,3,4),(3,4,5))$ introduces a mesh on top of these vertices. The flattened representation consists of three arrays $((x_0,x_1,x_2),(x_1,x_2,x_3),\ (x_2,x_3,x_4),\ \ldots)$, $((y_0,y_1,y_2),(y_1,y_2,y_3),\ (y_2,y_3,y_4),\ldots)$ and $((z_0,z_1,z_2),(z_1,z_2,z_3),\ (z_2,z_3,z_4),\ldots)$.

In the flattened representation, a mesh is converted into a sole stream of double coordi-

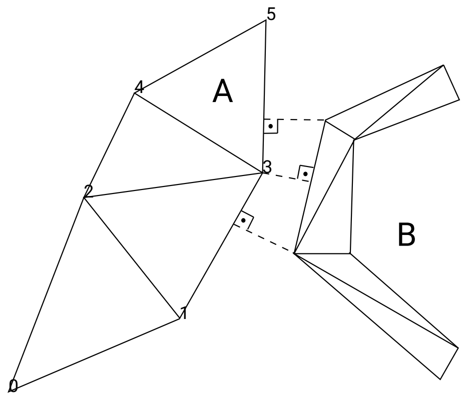


Figure 1: Two meshes A and B are checked against each other. An enumeration for the vertices of A is given. We denote the three shortest distances.

nates. The vertices are not indexed but replicated, i.e. the mesh topology is lost. If we however compare a point to the mesh, we basically can run through the flattened representation in one rush. There's no lookup required anymore.

A *contact point* in Δ is a point in-between two meshes which is closer to both of them than a given ϵ. Δ provides various contact identification routines in a namespace `delta::contactdetection`. They are passed the flattened geometry representations plus the ϵ, and they return a C++ vector of contact points. The latter are instances of a class `delta::ContactPoint`, and each instance represents its point in space plus a normal which points to the closest geometry. Contact points always are right in the middle of two geometries, i.e. if we invert the normal, we make it point to the other geometry. The normal by construction is smaller than ϵ. Each contact point furthermore holds a marker that identifies whether it is inside the two geometries or in-between. If this marker is set, the two meshes do overlap.

Example: Duplicated/close contact points (Figure 1) Contact points are derived on a per-triangle-pair basis. Let the three shortest distances from the figure denote the only triangle combinations that are closer than two times a given ϵ. The result of a contact detection invocation is four contact points: When we check A's triangle 1,2,3 against the bottom triangle from B, the lowest distance/contact point is detected. The next check against a triangle from B yields the normal rooted in A's point 3. Exactly the same point is found again when we compare A's triangle 3,4,5 next.

On top of the contact point detection, Δ provides routines to filter out redundant contact points. The filter is a simple iterative sort: We pass in a set C_{in} of contact points plus a parameter $h > 0$. It returns a set C_{out} of contact points. The contact point with the smallest normal in C_{in} is contained in C_{out}. From hereon, we construct C_{out} running over C_{in} with ascending normal length: A contact point from C_{in} is made member of C_{out} if no point from C_{out} is closer then the given threshold h. Such a simple redundancy elimination is useful for a mesh with mesh size h for example. We obtain the closest contact point plus all other points that are not closer than h to this guy. Redundant points within the mesh thus are eliminated, but we keep multiple contact points for concave meshes.

Δ's core set of routines is supplemented by mesh stretching, scaling and rotation routines. We offer a small set of basic DEM potentials for dash spring-pot potentials [2, 4], as well as a set of primitive construction and mesh import routines.

3. Algorithms

3.1. Vectorised triangle comparison

Δ's contact point detection is solely based upon triangles. It does, in its current version, neither exploit any mesh topology/hierarchy nor identify higher-dimensional contact manifolds. We thus assume, for the following discussion, that two sets of triangles $\mathbb{T}_1$ and $\mathbb{T}_2$ are to be compared, and the algorithm compares all triangles from the former to the latter. Its complexity is $\mathcal{O}(|\mathbb{T}_1| \cdot |\mathbb{T}_2|)$.

Bounding sphere comparisons Our code supports a range of comparison routines. The simplest type of comparison is the check of two spheres against each other. While this comparison is sufficient for solely sphere-based geometry models as they are predominant in DEM, e.g., we emphasise another important role of the sphere comparisons: Comparisons of triangulated objects can be very costly. It is thus reasonable to wrap them into a sphere pre-check. If the bounding spheres of two objects do not overlap, we can skip all further triangle comparisons.

Comparison-based The straightforward approach to compare two triangles is based upon some simple geometric considerations: The closest point between two triangles is either defined by a point from one triangle and a face of the other triangle, or it is found between the edges of two triangles. Once all distances between these geometry combinations are known, we can construct the closest distance by a simple minimum selection. If this distance is smaller than 2ϵ, two triangles yield a contact point by our definition.

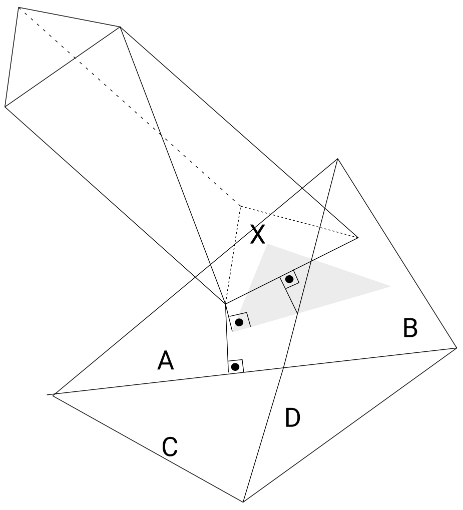


Figure 2: Geometric checks for a triangulated volume approaching a plane (at the bottom) spanned by four triangles A,B,C,D. The comparison of triangle X to A finds the contact point (shortest line) through a projection of a vertex of X onto A. Comparing X to C first obtains the same point projecting the vertex onto the plane spanned by C, but then it has to move the anchor point on the plane into C. When we compare X to B, the closest line is found between two edges.

The implementation of this algorithm in Δ is a simple sequence of geometric comparisons. We compare each point of the two triangles against the other triangle. This gives us $2 \cdot 3$ combinations. For the comparison of the three edges of the two triangles, we obtain $3 \cdot 2 \cdot 1$ combinations to check. We exploit symmetries. The number of geometric setups that are to be compared totals to 15 (Figure 2).

While a constant complexity/variant count per se is nothing bad, we note that the individual comparisons require additional checks. Whenever we project a vertex from one triangle onto the plane spanned by the triangle counterpart, we next have to check whether an orthogonal projection ends up within the triangle. Otherwise, the closest point-to-triangle point hits an edge. Analogous considerations holds for edges.

The code is heavy on if statements. We study its vectorisation behaviour in [4] and demonstrate that it struggles to vectorise and to stream data through the core. However, this code variant always yields the exact result. It is robust.

An iterative approach An alternative approach casts both comparison triangles into their barycentric representation defined by four scalars $a_1, a_2, a_3, a_4 \in [0,1]$. Let the triangles be written as points $T_1(a_1,a_2), T_2(a_3,a_4) \in \mathbf{R}^3$ in the three-dimensional space. The closest point between the two triangles is given as

$$\min_{a_1,a_2,a_3,a_4} \frac{1}{2} |T_1(a_1,a_2) - T_2(a_3,a_4)|_2^2 \quad \text{subject to } a,b,c,d \in [0,1].$$

We cast this minimisation problem with a Lagrangian $\alpha_L > 0$ multiplier into a weak formulation

$$\min_{a_1,a_2,a_3,a_4} \frac{1}{2} |T_1(a_1,a_2) - T_2(a_3,a_4))|_2^2 + \alpha_L \sum_i \max(a_i - 1, 0) + \max(-a_i, 0) \qquad (1)$$

and solve it via a Newton method. The system can become notoriously ill-conditioned for close-to-parallel triangles and thus requires some regularisation of the underlying Jacobian. We add a diagonal matrix scaled with an additional regularisation weight α_R. All matrix inverts from (1) then can be written down explicitly as we are solely working in the three-dimensional space. With max using simple masking within the chip, the code almost lacks ifs.

We obtain an iterative algorithm with high arithmetic intensity—notably if we run the Newton for multiple triangles in one rush. Its problem is that it is not robust even though the majority of triangle-to-triangle comparisons converges after less than four iterations.

A hybrid approach It is easy for (1) to track the residual per step. The resulting Newton update size is a reasonable indicator for convergence if Lagrangian and regularisation parameters α_L, α_R are well-chosen. If this error estimator is big after a fixed number of steps, we may assume that the iterative scheme has not yet converged and is potentially ill-conditioned.

Empirical studies show that the iterative scheme is by more than a factor of 10 faster then the geometric checks, if we constrain the number of Newton steps to 4–8. Δ's hybrid approach thus runs a prescribed number of Newton steps. Hardcoding the iterations has the nice side effect that we can unroll these steps. After the fixed number of iterations, it checks the termination criterion. If it exceeds a prescribed error threshold, the code falls back to the pure geometric checks.

We end up with a geometric comparison duet which is empirically by a factor of two slower than a purely iterative approach, but, different to this one, is absolutely stable; and still more than a factor of 5 faster than an if-based only approach [2, 4]. The iterative approach is computationally more demanding than its geometric counterpart. The gain in speed is thus solely due to vectorisation which renders this approach promising for state-of-the-art architectures.

3.2. Multicore parallelisation

Geometry-based solvers typically rely on a powerful spatial decomposition to obtain parallel code. It nevertheless is important for a geometry library to introduce further parallelism that notably is lightweight and thus targets multicores. Δ's parallelisation relies on OpenMP though we note that it is straightforward to combine OpenMP with Intel's Threading Building Blocks [5] or further alternatives.

Intra-library parallelisation Within Δ, we provide both routines to compare individual triangles with each other as well as routines which compare whole triangle sets $\mathbb{T}_1$ and $\mathbb{T}_2$ to each other.

A straightforward parallel realisation decomposes the input data: We compare the first triangle from $\mathbb{T}_1$ against all triangles from $\mathbb{T}_2$, while we concurrently check the second triangle against all counterparts, and so forth. Dynamic scheduling allows the runtime to react to load imbalances which arise from triangle counts that are not multiples of the cores as well as from the hybrid geometry solves: If geometric re-checks become necessary for a comparison, the affected comparison is more expensive than a counterpart where the Newton solve has terminated successfully.

Experimental evidence on KNL and Broadwell architectures suggests that the intra-library parallelisation starts to upscale once the individual mesh consists of 40–80 triangles. For geometries with lower resolution, the intra-library parallelisation cannot yield a observable speedup.

Functional realisation and further parallelisation approaches All building blocks in Δ that have been discussed for far are realised without side effects: There are no global states in the library. As a result, the routines are thread-safe. In this context, we refer notably to [2] where we use the routines that inspired Δ within a multiscale domain decomposition and obtain reasonable scaling even for dynamically adaptive meshes.

3.3. Spatial caching

The vertices of meshes often exhibit some spatial compactness. If we study particles, they typically do not span the whole computational domain. Codes that sample a geometry over a domain typically do not traverse this domain randomly. They do not jump around. As a result, we may assume that many consecutive geometric queries arise from within a certain subdomain.

If a triangle or point is checked against a mesh, and we find out that the distance between the two of them is larger than an $\epsilon + \delta$, then a subsequent query for a triangle or point which is within a δ area are around the first triangle can be answered straight away: there is not contact point.

To exploit this observation, we enable Δ to build up an octree. Each octree's node holds a minimal distance scalar to the actual mesh. The octree's leaves then point to the actual triangles of the mesh. For geometric queries, we traverse the octree top-down, i.e. from coarse to fine resolution and validate if the query can be answered straight away with a “there is no contact”. We use the spatial layout of input and comparison mesh to construct this octree on-the-fly. If the octree suggests that there might be contact, we assemble the contact result bottom-up from the tree: Our code identifies all octree nodes which might link to mesh triangles yielding collision points, runs the comparisons per node, and then concatenates all results. We can read the octree approach as an adaptive bounding box pre-check.

If codes hold many meshes (geometric) objects, Δ holds one octree per mesh. If codes run a domain decomposition, the octree construction implicitly follows this domain decomposition. If a mesh is modified, we found it the easiest to throw away the octree and to reconstruct it from scratch following any follow-up geometric queries. We note that the octree construction is, per definition, not functional. Instead of a sophisticated semaphore concept, we thus provide a veto mechanism for on-the-fly octree refinement and an a priori octree construction function.

4. Demonstrators

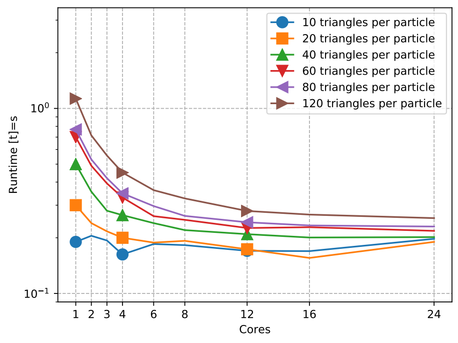
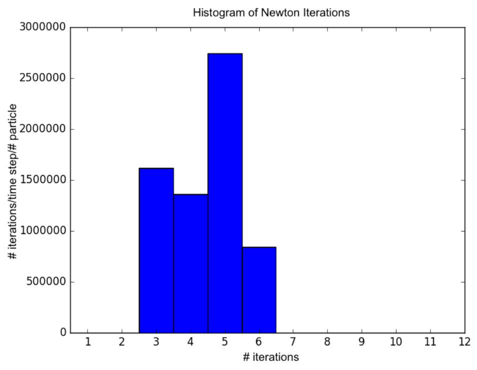


Figure 3: Scalability of the triangle-to-triangle comparisons in a DEM context where two particles collide. From [2].

Intra-library parallelisation studies In [2], we compare two particles colliding with each other. The collision is based upon a spring-dashpot potential. Insights from this paper have been directly transferred into the Δ library. We summarise that the library has a working internal parallelisation. However, this parallelisation yields reasonable speedup if and only if the individual meshes (particles here) are of reasonable triangle count (Figure 3). The library can become an HPC building block. It does not yield scalability per se. The same paper validates that the library can be used within a multithreaded environment. If Δ's internal scalability is combined with an additional, outer level of parallelism, we have demonstrated excellent scalability on up to 24 cores on Broadwell and 32 cores on Intel Knights Landing (KNL).

Table 1: Hardware counter results for characteristic single-core runs on the Sandy Bridge chip. BF means brute force. Table from [4].

Metric	Stream	BF	Penalty	BF+SIMD	Penalty+SIMD
Runtime (s)	6.71	22.49	15.47	7.78	4.54
MFLOPS/s	1,245	962	1,073	2,808	3,202
CPI	0.48	0.48	0.49	0.91	1.26
Bandwidth MB/s	14,120	408	579	902	1,424



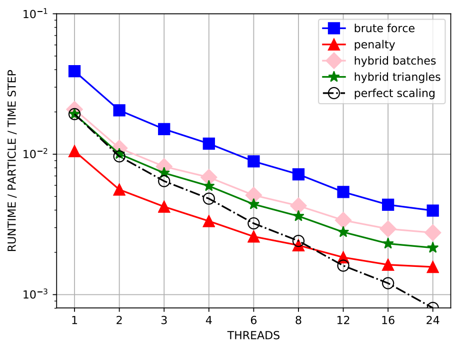


Figure 4: Left: Typical number of required Newton iterations if we equip the iterative scheme with a dynamic termination criterion. Right: Scalability of three core comparison approaches. From [4] and [2].

Single core performance studies Single core performance studies are published in [4]. On a Broadwell, the paper compares all data to the Stream TRIAD benchmark. While both the penalty and the geometric approach (named brute force as all potential geometric variants have to be checked) obtain a speedup from vectorisation, the penalty-based approach exhibits better streaming behaviour and obtains higher flop rates resulting in lower time-to-solution.

Algorithmic tuning The insight that a fixed, small number of Newton iterations is sufficient for most triangle configurations is validated in Figure 4; again for DEM. The outliers where triangles yield ill-conditioned Newton iterations are so few that one cannot see them in the plot. Nevertheless, a robust geometry library has to be prepared for them. Our hybrid robust scheme provides this and still exhibits advantageous scaling behaviour (Figure 4, too).

5. Conclusion, outlook, and about the code

Δ is a simple, small library for geometry handling as it is for example required for immersed boundary methods or Discrete Element Methods. The focus of this library is not an extensive set of features. We did write an HPC code building block. Experimental validation as well as further details about Δ's use in a DEM context is published in [2, 4].

About the code The code as well as all documentation are available through www.peano-framework.org or other URLs linking to the PI's software archive. A nightly build updates all source code documentation. It is realised through doxygen. While archive downloads are built each night, too, we provide free repository access for users who sign up. The actual software is hosted at the Leibniz Supercomputing Centre (LRZ) in a git repository.

We currently have two versions of the code: Our "legacy" code is the code with all the advanced features described in the present document. A new code base that's currently rolled out through the repository is a rewrite of this legacy/proof-of-concept software which focuses on clean, minimalist interfaces and simplicity.

Outlook and limitations Our current Δ release provides solely support for flat meshes, i.e. meshes without hierarchy. It is subject of current research how to exploit meshes which exhibit hierarchy. We also actively work on further extensions which do not rely on third-party discretisations of spline-based geometric models.

The code at the moment exclusively supports point contact. Region overlaps are not identified by the code base. This is subject of further work, too. We finally note that Δ has no inherent MPI support. We rely on the the idea that the meshes either are held globally on each rank or that the partitioning and mesh distribution is organised by the user. We do however build up our cache trees on-demand and thus follow any domain decomposition implicitly.

Acknowledgements

This manuscript contains content and ideas from Konstantinos Krestenitis' PhD thesis, which has been sponsored by EPSRC (Engineering and Physical Sciences Research Council) and EDF Energy as part of an ICASE studentship (award ref 1429338). We do appreciate EPSRC's support of this research through the eCSE12-4 programme (grant "Project Meshes—a HPC geometry library supporting mesh-to-mesh projections") organised through the EPCC and Archer. This work made use of the facilities of the Hamilton HPC Service of Durham University. All software freely available from authors' webpages [3].

References

[1] The Open-Asset Imports-Lib, last visited 01/10/2018. www.assimp.org.

[2] K. Krestenitis and T. Weinzierl. A multi-core ready discrete element method with triangles using dynamically adaptive multiscale grids. *Concurrency and Computation: Practice and Experience*, pages e4935:1–17, 2018.

[3] K. Krestenitis and T. Weinzierl. Δ—a fast geometry library for tessellated geometries, last visited 01/10/2018. hpcsoftware.dur.ac.uk or www.peano-framework.org.

[4] K. Krestenitis, T. Weinzierl, and T. Koziara. Fast DEM collision checks on multicore nodes. In R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, editors, *Parallel Processing and Applied Mathematics*, volume 10777 of *Lecture Notes in Computer Science*, pages 123–132, 2018.

[5] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[6] M. Tavelli, M. Dumbser, D. E. Charrier, L. Rannabauer, T. Weinzierl, and M. Bader. A simple diffuse interface approach on adaptive cartesian grids for the linear elastic wave equations with complex topography. *arXiv:1804.09491*, 2018.