Volume xx (200y), Number z, pp. 1–14

Real-Time Fluid Effects on Surfaces using the Closest Point Method

S. Auer[†,1], C. B. Macdonald[‡,2], M. Treib[§,1], J. Schneider[¶,3], R. Westermann[‖,1]

[1]Technische Universität München
[2]Oxford Centre for Collaborative Applied Mathematics (OCCAM)
[3]King Abdullah University of Science and Technology

Abstract

The Closest Point Method (CPM) is a method for numerically solving partial differential equations (PDEs) on arbitrary surfaces, independent of the existence of a surface parametrization. The CPM uses a closest point representation of the surface, to solve the unmodified Cartesian version of a surface PDE in a 3D volume embedding, using simple and well-understood techniques. In this paper we present the numerical solution of the wave equation and the incompressible Navier-Stokes equations on surfaces via the CPM, and we demonstrate surface appearance and shape variations in real-time using this method. To fully exploit the potential of the CPM, we present a novel GPU realization of the entire CPM pipeline. We propose a surface-embedding adaptive 3D spatial grid for efficient representation of the surface, and present a high-performance approach using CUDA for converting surfaces given by triangulations into this representation. For real-time performance, CUDA is also used for the numerical procedures of the CPM. For rendering the surface (and the PDE solution) directly from the closest point representation without the need to reconstruct a triangulated surface, we present a GPU ray-casting method that works on the adaptive 3D grid.

Categories and Subject Descriptors (according to ACM CCS): I.6.8 [Simulation and Modeling]: Types of Simulation—Parallel I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Techniques for solving partial differential equations (PDE) have numerous applications in physics-based simulation and modeling, and they are frequently employed in computer graphics for realistically simulating real-world phenomena such as fluids or deformable solids. Most commonly, partial differential equations over $\mathbb{R}^2$ or $\mathbb{R}^3$ are considered, and, since analytical solutions only rarely exist, numerical solutions using some form of spatial discretization are employed.

In computer graphics, numerical solutions of PDEs are also used to produce effects on manifolds such as surfaces in $\mathbb{R}^3$. Prominent examples include surface texture synthesis [Tur91], texture assignment [CLB*09], or flow simulation on surfaces [Sta03]. If an isometric surface parametrization exists, a PDE defined on a surface can be transformed into a PDE on the 2D parameter domain and solved using standard discretizations [Sta03, LWC05, LFW07]. PDEs on surfaces can also be solved directly using finite element discretizations on a surface triangulation [DE07]. These techniques, however, rely on changing the non-parametric form of the PDE to agree with the underlying discretization. An alternative are so-called embedding techniques [BCOS01, NNRW09, CLB*09, RM08, CRT04], which lift the PDE to a narrow band around the surface and solve a transformed PDE on this band.

Embedding techniques are particularly attractive because they do not rely on the existence of a low distortion parametrization. The computation of such a parametrization can be difficult to achieve, if not impossible, and it is not suitable in applications where the surface undergoes frequent

† e-mail:auer@in.tum.de
‡ e-mail:macdonald@maths.ox.ac.uk
§ e-mail:treib@tum.de
¶ e-mail:jens.schneider@kaust.edu.sa
‖ e-mail:westermann@tum.de

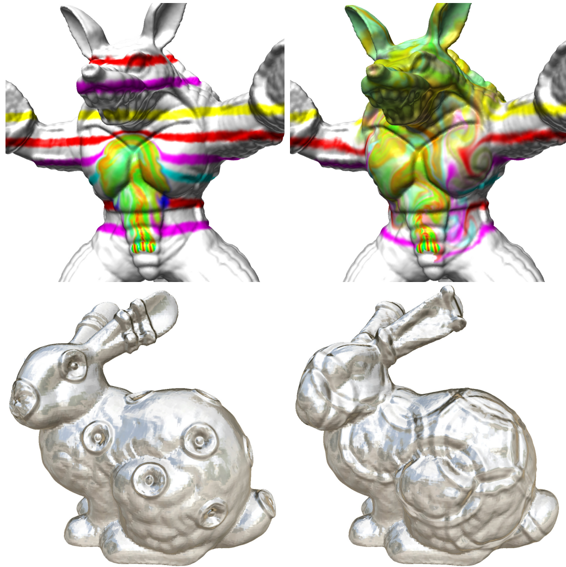

Figure 1: *Numerical simulation of fluids on complicated surfaces via the Closest Point Method. Top: The simulation result is rendered via rasterization and pixel shading. Bottom: Raycasting the embedding computational grid allows simulating surface displacements. At a resolution that corresponds to a* 512^3 *Cartesian grid, simulation and rendering takes less than 160 ms per time step.*

shape changes. Another advantage of embedding techniques is that once the PDE has been lifted to the embedding 3D space, standard numerical schemes can be used in this space to efficiently solve the PDE.

Among the embedding techniques, the Closest Point Method (CPM) is one of the less complicated methods in that it only requires the existence of a closest point representation of the surface (i.e., for any point in the embedding space, the surface point with the least Euclidean distance is known) and it solves completely standard PDEs—without any metric terms—in the embedding space using common numerical methods on a uniform Cartesian grid [RM08]. In addition, the CPM can handle general open surfaces and surfaces without orientation, and, thus, is applicable even in scenarios where the surface separates into parts.

Despite its simplicity and accuracy, however, a naïve implementation of the CPM is exhaustive in memory. Such an implementation would typically construct a full surface-embedding Cartesian grid and pre-compute at any grid point its closest surface point. It is clear that this severely limits the resolution of the computational grid to be used. Furthermore, the closest point representation remains fixed only under the assumption that the surface does not change. In scenarios where this is not the case, re-computing closest points can have a severe impact on performance.

Our contribution. The primary focus of this paper is the simulation of fluid effects on surfaces in real time using the CPM. To achieve this, we present a fast GPU method for realizing the CPM *and* rendering the simulation results. The method can efficiently solve non-linear PDEs on surfaces using a high-resolution embedding grid, and it can calculate and render the resulting fields independently of the input surface resolution. Due to the strict separation of the surface representation and the computational grid, our method supports initial surfaces of arbitrary topology and geometric detail. The method is implemented in CUDA and Direct3D10, and all parts have been optimized to make the best possible use of the massive parallel processing power and memory bandwidth on current GPUs.

The particular contributions of our paper are:

- A novel CUDA method for constructing an adaptive multiblock closest point grid representing a narrow spatial band around a given triangulated surface.
- A CPM for numerically solving the wave equation and the Navier-Stokes equations on arbitrary surfaces.
- A GPU ray-caster that works directly on the adaptive closest point grid and can visualize surface appearance attributes and surface displacements.

Our paper is structured as follows: In Section 2 we discuss previous work that is related to ours. The CPM is reviewed in Section 3. Section 4 is dedicated to the efficient construction of an adaptive computational grid enclosing a triangulated surface representation, and the generation of the closest point representation. Section 5 sheds light on the numerical solution of the second-order linear wave equation and the incompressible Navier-Stokes equations via the CPM. Besides discretization aspects, the effects of the closest point extension on stability, accuracy and performance of the simulation are discussed. Section 6 introduces the volume raycaster that is used to visualize dynamic color and displacement effects independently of the surface triangulation. Section 7 presents a detailed performance analysis. The paper is concluded with a discussion of the advantages and limitations of the proposed method and some ideas for future work.

2. Related Work

An interesting surface-based problem is the simulation of fluid dynamics on surfaces. Stam applied a modified 2D version of his stable fluid dynamics simulation [Sta99] to the patches of Catmull–Clark surfaces and introduced a technique to transfer information in an overlap between the patches [Sta03]. While working well in regular areas of the surface, the approach leads to errors at non-valence-four vertices. For techniques relying on some sort of global parametrization similar problems can be expected at the singularities.

Several other authors directly use the vertices and edges of triangle meshes as a discretization and employ finite

element, finite volume, Lattice Boltzmann or related numerical methods in their solvers [SY04, FZKH05, NMZ07, ATBG08]. This however takes away some advantages of a regular grid discretization as finite element methods are generally more complicated. Particularly on surfaces, the resulting algorithms may be quite difficult to implement in practice, e.g. [RWP06]. Finally, if the input surface is not given as a triangulation a costly reconstruction procedure may be required.

The interest in simulation on surfaces also resulted in particle based solutions. Turk [Tur91] simulated reaction diffusion systems in a grid formed by particle relaxation on a surface, with connectivity given by a Voronoi diagram. Later, the technique was adapted to general shallow water equations [WMT07]. Bürger et al. [BKW10] use an orthogonal fragment buffer to trace particles along a surface in order to color the surface.

Chuang and colleagues [CLB*09] proposed a volume embedding scheme based on B-splines as Ansatz-functions in a hexahedral simulation grid. By restricting the basis functions to a surface instead of a sub-volume, a weak form of a surface PDE can be solved independently of the 3D simulation domain, but at the expense of explicitly clipping the surface mesh against the simulation cells.

The Closest Point Method was introduced by Ruuth and Merriman [RM08]. Later works presented an implicit time stepping [MR09] and used the CPM for the evolution of level sets [MR08] or segmentation on surfaces [TMR09]. Hong et. al [HZQW10] applied the CPM to fire simulation on animated surfaces. Due to the embedding used, the CPM relies on certain properties of distance transforms, such as smoothness of properties close to the surface. Jones [JBS06] provides an excellent survey of these properties.

An additional topic related to our method is GPU-based surface voxelization of triangle meshes. We borrow ideas from several authors [DCB*04,ED06,ZCEP07,SS10,Pan11] and perform a surface voxelization to determine all voxels lying within a computational band of arbitrary thickness around the surface. For these voxels, the CPM surface representation stores the closest points on the surface. As such, our technique computes a partial distance volume around the surface, including the positions of the closest points. Building upon the work of Pantaleoni, Schwarz and Seidel [SS10, Pan11], we demonstrate the efficient use of the CUDA parallel programming API for constructing a multi-block closest point grid.

3. The Closest Point Method

As mentioned above, the CPM is an embedding method where computations are performed in an embedding space surrounding the surface (typically a 3D volume) in such a way that the results are consistent with the solution of a surface partial differential equation. The main principle needed for this to be true is that of "equivalence of gradients": the intrinsic surface gradient $\nabla_S u$ of a surface function u agrees *on the surface* with the standard Cartesian gradient of a volume function v, provided v is the closest point extension of u [RM08]. Intuitively this is because the closest point extension $v(\mathbf{x}) = u(cp(\mathbf{x}))$ is constant in the normal direction to the surface so the change in v must be tangent to the surface. Here, $cp(\mathbf{x})$ denotes the surface point closest to the point $\mathbf{x}$. A second principle applies in a similar fashion to surface divergence operators [RM08]. The two principles can be combined to handle many other differential operators including the Laplace–Beltrami operator.

To build a numerical method on these ideas, consider a prototype problem describing the evolution of some attribute u on a surface

$$u_t = f(\nabla_S u)$$

where f is a general nonlinear function and $\nabla_S u$ is the intrinsic surface gradient. Suppose at some fixed time t we have a solution v^t defined over the volume which is constant in the normal direction to the surface. We can move the solution forward in time by one step of size Δt using, for example, a forward Euler discretization. This *evolution phase* requires us to evaluate the right-hand side at time t and by the principles mentioned above, this is exactly the same (for points on the surface) as evaluating $f(\nabla v^t)$ in the volume:

$$\tilde{v}^{t+1} := v^t + \Delta t f(\nabla v^t).$$

The new solution $\tilde{v}^{t+1}$ might not be constant in the normal direction so we then perform a second *extension phase*, a closest point extension which projects values off the surface into the surrounding volume to obtain a solution v^{t+1} which is constant in the normal direction:

$$v^{t+1}(\mathbf{x}) := \tilde{v}^{t+1}(cp(\mathbf{x})), \quad \text{for each point } \mathbf{x}.$$

To obtain values at $cp(\mathbf{x})$, which does not have to coincide with a grid sample $\mathbf{x}_G$ in the discrete case, higher-order interpolation is used. We then repeat these two phases over and over to advance the solution in time. Several properties make the Closest Point Method simple and effective:

- In the embedding volume, simple well-understood finite difference schemes can be applied to evaluate $f(\nabla v)$.
- Posing the PDE in the embedding volume is easy: in our example, the nonlinear function f is *unchanged* in the volume calcuation and we simply replace intrinsic differential operators with their standard Cartesian counterparts. There are no metric terms to deal with.
- In the extension phase, we need the closest point for each grid point in our embedding volume. This is the only geometry representation of the surface that is needed and it is a very general representation: allowing arbitrary codimension and non-orientable surfaces for example.
- Without effects on the accuracy, the computation can be performed on a narrow band enveloping the surface. Due

to the closest point extension, artificial boundary conditions do not need to be applied at the band's exterior.
- The accuracy of the method is well-understood and depends on the accuracy of the time-stepping scheme, the finite difference scheme and the interpolation scheme.

In principle, the discrete closest point surface representation required by the CPM is a volumetric Cartesian grid, where each grid cell stores the surface point closest to its center. When the surface is smooth, the closest point is unique in a small band around the surface. However, in our application where the CPM is used to simulate effects on surfaces given by piecewise linear triangle meshes, the surface is only $\mathcal{C}^0$ at the edges. Thus, discontinuities in the closest point field can arise, and for a single grid point more than one closest point can exist on the surface. In this case we select one of these closest points arbitrarily. The closest point extension propagates such discontinuities also to the fields which store the simulation attributes. Both the extension phase and the evolution phase must therefore employ numerical schemes that are tolerant of discontinuities.

3.1. Adaptive Multiblock CPM

For high resolution simulations, the use of a uniform simulation grid is impractical. For the CPM, on the other hand, it is sufficient to work on two narrow computational bands around the surface: The *evolution band* and the *extension band*, which are used in the evolution and extension phase of the CPM, respectively. This suggests using an adaptive multiblock grid, which stores only those blocks of the full grid which overlap these bands. Figure 2 illustrates such a multiblock grid in 2D. The widths we choose for the bands are the ones proposed by Macdonald and Ruuth [MR08] (Section 4.1.1).

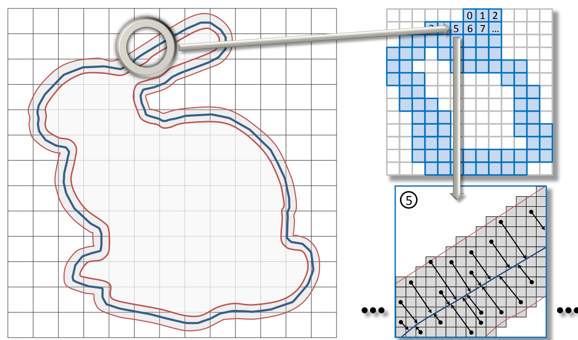

Figure 2: *Sparse closest point grid. Left: A grid of blocks, some of them overlapping the narrow band around the surface. Top right: Only blocks overlapping the narrow band around the surface are stored in linear order. Bottom right: The closest points of the grid cells within these blocks.*

The multiblock grid is comprised of two levels. The coarse level consists of a regular 3D grid, where each cell represents a block of cells in the uniform simulation grid. The fine level consists of the grid cells of those blocks which are intersected by the computational band. All data is stored in linear buffers. The coarse-level cells representing an intersected block store the position in the fine level buffer at which this block is laid out. For constructing a closest point surface representation we first compute the blocks which are intersected by the computational band. This is performed by determining for every triangle separately the blocks which are intersected by the band around it. The cells of a block determined in this way are stored in a contiguous region in the fine-level buffer. To allow for an efficient computation of the closest point of each fine-level cell, we also compute the set of triangles intersecting this block. For each cell of the same block we then iterate over the corresponding set of triangles and compute the closest surface point. Finally, the simulation is carried out on the fine-level cells.

4. CPM on the GPU

Since we are aiming to support both static and time-varying surfaces, for instance surfaces that are modified in turn by the CPM simulation, an important requirement is that the creation and update of the sparse closest point representation can be performed at high rates even for high-resolution surfaces and simulation grids. For this computationally and bandwidth intense task, we exploit the massively parallel architecture of the GPU.

Our proposed algorithm for constructing a closest point volume is inspired by recent CUDA voxelization approaches [SS10,Pan11]. Similar to the tile-based approaches proposed in these works, we use a sort-middle rasterization pipeline which first assigns triangles to coarse blocks and then performs a fine-grain voxelization and closest point calculation per block. There are, however, some specific differences of our GPU algorithm:

- It computes a surface voxelization using a distance criterion that is based on the thickness of the computational band around a surface.
- It calculates for every voxel within the computational band its closest surface point and the distance to this point.
- It constructs an adaptive 2-level multiblock grid designed for efficient simulation.
- It parallelizes over fine level cells instead of triangles to avoid shared memory atomics in the closest point calculation.

4.1. Data Layout and Grid Generation

For the sake of clarity, we will first describe the GPU construction of the sparse multiblock grid, including the indexing schemes that are required to efficiently perform the CPM, before we go into the details of the actual closest point calculation.

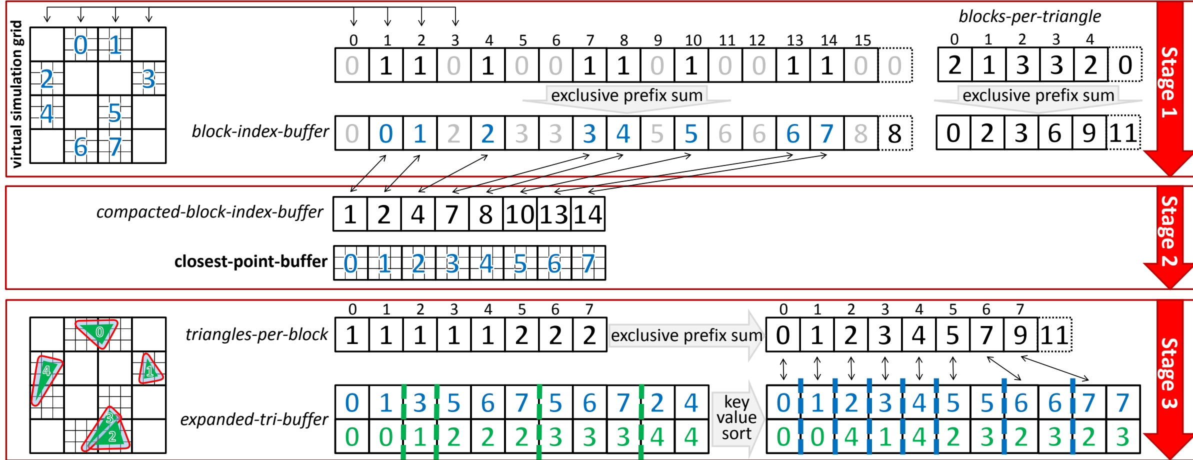

Figure 3: *Overview of the different GPU buffers that are created to facilitate an efficient realization of the CPM.*

To create the sparse simulation grid, the user first selects a simulation resolution by specifying the size $\Delta\mathbf{x}$ of one simulation cell. Then the bounding volume of the triangle mesh—enlarged to respect the width of the computational band—is subdivided into $k \times l \times m$ equally sized blocks. We assume that each block contains b^3 simulation cells, so that the values of k, l, and m can be computed.

The grid construction on the GPU consists of three stages: First, we determine which and how many blocks are intersected by the computational band. The *block-index-buffer* is used to keep track of this information. Second, the *closest-point-buffer*—large enough to store the closest points of all simulation cells represented by these blocks—is allocated. To accommodate fast access to cells in adjacent blocks, an additional buffer—the *compacted-block-index-buffer*—is created. It has as many entries as there are intersected blocks, and it stores the positions of these blocks in the *block-index-buffer*. From this, and by taking into account that in the *block-index-buffer* all blocks are laid out in x/y/z-order, adjacency information can be resolved. Third, the *expanded-tri-buffer* is constructed to support coalesced memory write operations in the closest point construction (see the following subsection). This buffer stores, in contiguous sections, the sets of triangles that are required by each block to compute the closest point grid. The different buffers are illustrated in Figure 3. In the following, we will describe the parallel CUDA implementation of the three stages.

Stage 1: One CUDA-thread is launched per triangle, and each thread determines the blocks that are intersected by the computational band around this triangle. Therefore, each thread computes the triangle's axis aligned bounding box—enlarged by the width of the computational band—and computes the blocks that are intersected by this box. Since testing against the bounding box can result in false positives, i.e., blocks that are intersected by the bounding box but not the computational band, an additional test of the blocks is performed to prune as many of them as possible: For every intersected block, the thread calculates the distance from the block center to the triangle and skips the block if its bounding sphere does not intersect the triangle's computational band. By writing a 1 into *block-index-buffer* at the positions of the remaining blocks, these blocks are marked as intersected. Because all threads write the same value, a synchronization of the write operations is not necessary in this case.

At the end of stage 1, each thread writes the number indicating how many blocks are intersected by the triangle to a temporary buffer—*blocks-per-triangle*—in global GPU memory. Finally, an exclusive parallel prefix sum [Har07] is computed in-place over *block-index-buffer* and *blocks-per-triangle*. At the positions that were marked, the *block-index-buffer* now contains the relative positions of the respective block in the sequence of marked blocks.

Stage 2: Since the prefix sum operation also calculates the total number of marked blocks, the *closest-point-buffer* can be allocated in GPU memory. Then, a CUDA kernel with one thread per entry in *block-index-buffer* is executed. The ith thread reads the index value *val* from the buffer at position i and compares it to the value that is stored at position $i+1$. If the values are different, the thread writes the value i into the *compacted-block-index-buffer* at position *val*.

Stage 3: We start by allocating the *expanded-tri-buffer* in GPU global memory using the size indicated by the prefix sum over *blocks-per-triangle*. The elements of this buffer are pairs of block-triangle indices, and for each triangle these pairs are written in succession into the buffer. Since the parallel scan operation also gives the starting positions of each set of block-triangle pairs in *expanded-tri-buffer*, a CUDA kernel with one thread per triangle is executed to first build these sets in shared memory—including the computation of intersected blocks as described before—and then to write them into the buffer at the starting positions (see the first

occurrence of *expanded-tri-buffer* in Figure 3). In the same kernel, we use CUDA's *atomicAdd* operation to fill a buffer *triangles-per-block*, which stores for every marked block the number of triangles it intersects.

In a second pass, *expanded-tri-buffer* is sorted with respect to the block ID using the CUDA radix sort [Har07], and a prefix sum over *triangles-per-block* is calculated. From the content of the resulting buffers the indices of all triangles contributing to the closest point representation for a particular block can be determined.

4.2. Closest Point Computation

For each of the cells in the *closest-point-buffer*, all of which are uniquely defined by the block they belong to and their relative position in the sequence of cells of this block, the point on the surface that is closest to the cell's center has to be computed (we will call these centers the grid vertices from now on). The parallel CUDA implementation we propose is optimized to reduce global memory operations by performing a block-wise closest point computation in shared memory, and writing the results en-block into global memory.

The CUDA kernel to compute the closest point representation executes one warp of CUDA threads per grid block. Every thread computes the closest point of exactly one grid vertex, by sequentially traversing the triangles that are stored in *expanded-tri-buffer* for that block. Thus, the threads of one warp process a number of grid vertices in parallel, in the same order they lie in memory, meaning that all required global memory buffers are accessed only by coalesced, fully utilized memory transactions. Furthermore, since the threads of a warp are executed in lock-step and every grid vertex is processed by exactly one thread, all synchronization and atomic instructions can be omitted.

In shared memory, two buffers are allocated to store the closest point coordinates and the distances of the grid vertices to these points. All threads of one warp first perform a coalesced global memory read operation to fetch the triangles from *expanded-tri-buffer* that are needed to compute the closest points. The number of triangles is read from *triangles-per-block*.

Each thread computes the grid vertex for the cell it is working on. Then, it iterates over all triangles, and for each of them it computes the closest point and corresponding distance. If the distance is shorter than the one stored in the shared memory buffer, both the distance and the closest point coordinate are updated. The closest point computation is performed by first determining whether the grid vertex is closest to a corner, an edge, or the triangle's interior. By a coordinate system transformation that brings the triangle into one of the coordinate planes, this essentially breaks down to a 2D problem: According to the illustration in Figure 4, we need to identify in which of the 7 zones of the triangle the projected vertex $\mathbf{x}_0$ lies. After all triangles have been processed, the shared memory buffers are copied in coalesced, fully utilized memory transactions to buffers in global memory.

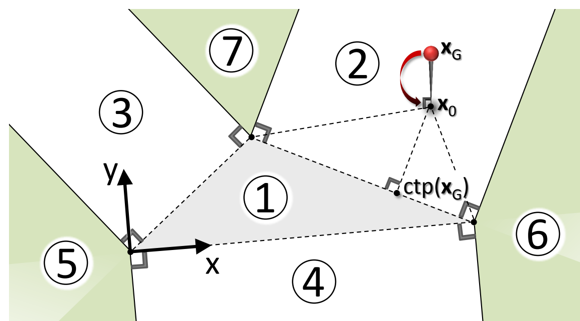

Figure 4: *In each of the 7 zones, $\mathbf{x}_G$ is closest to one region of the triangle (face, edge, corner).*

5. Fluid Simulation

In the following we describe the simulation of fluid effects on surfaces using the CPM on the GPU. We focus on the numerical solution of two different PDEs describing such effects: the wave equation and the incompressible isothermal Navier-Stokes equations.

Before a PDE solution can be computed via the CPM, appropriate initial conditions have to be specified (Section 5.1). Then, each solution step generally consists of two phases, both implemented on the GPU via CUDA: In the evolution phase, the solution of the embedding PDE (Section 5.2) in the next time step is calculated for all grid vertices in the evolution band. After the evolution, the closest point extension (Section 5.3) is applied to all vertices in the extension band. In this phase, the solution is propagated from the surface to the computational band by replacing the values at the grid vertices with values interpolated at the closest points on the surface.

For the wave equation (Section 5.4), the less involved of the two presented fluid models, just one evolution and one extension is necessary for each time step. In contrast, for the incompressible Navier-Stokes model (Section 5.5) a splitting method was used to handle the terms of the equations independently of each other. Consequently, in this model a time step requires several sub-steps and also more than one closest point extension (see Figure 5).

5.1. Initialization

The definition of the computational bands and the construction of the closest point representation (Section 4) are necessary to start the CPM. The initialization of the solution variable completes the initialization phase. In this step, initial data on the surface has to be extended to the simulation

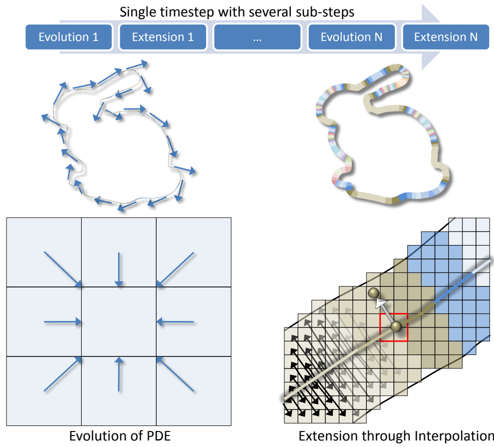

Figure 5: *A time step may require several sub-steps, consisting of an evolution (left) and an extension phase (right).*

grid. When the surface is given as a triangle mesh, the initial data is typically given as per-vertex attributes or as a texture map. Every grid vertex in the extension band interpolates the surface data at the closest point and stores this value at its location in the simulation grid.

5.2. Evolving the PDE

To update the simulation attributes with the solution for the next time step, we run one CUDA thread per grid vertex of the intersected blocks. If the vertex is not within the evolution band, the thread terminates immediately. For each term of the embedding PDE we choose an adequate evolution strategy (see Sections 5.4 and 5.5). The embedding of terms of the PDE involving spatial differential operators up to order two is found by replacing surface gradients, surface divergences and Laplace–Beltrami operators with their standard Cartesian counterparts in $\mathbb{R}^3$. The discretization can then be done via regular finite differences. The appealing property of CPM in this case is that information is automatically propagated only in tangential directions along the surface. When we have an existing discretization for the problem of choice in $\mathbb{R}^3$, we can reuse it without modification.

The user steers the interactive simulation at runtime, by modifying the simulation attributes temporarily or placing permanent Dirichlet boundary conditions. When this happens, we find the first surface point under the mouse cursor via raycasting (see Section 6.1) where we then attach a sphere with a radius equal to the width of the extension band. In every time step, one CUDA thread is started for each grid vertex within one of these spheres in order to replace the respective attribute values.

5.3. Closest Point Extension

The closest point extension is performed by one CUDA thread per grid point in the extension band. We retrieve the grid point's closest point on the surface and resample the respective attribute at this position using an interpolation scheme. The value at the grid point is then replaced by the interpolated value, thus creating a field that is constant in normal directions to the surface. For values outside the evolution band this can be seen as a natural form of Neumann boundary condition, because gradients in directions normal to the boundary are always zero. The kind of interpolation is important and influences stability, quality and performance of the whole simulation.

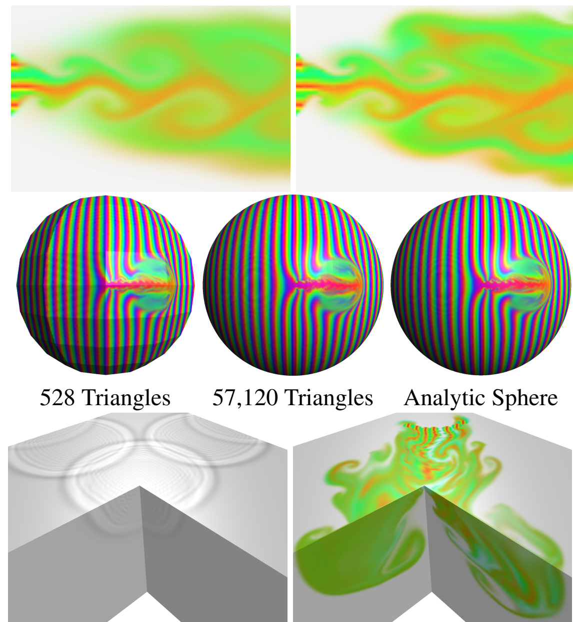

Figure 6: *Top: With linear interpolation (left) instead of WENO4 interpolation (right) the simulation shows artificial diffusion and loses energy. Middle: Initialization with a high-quality triangle mesh instead of an analytic surface leads only to subtle differences in the simulation results (grid size: 256^3). Bottom: While the CPM generally requires the surface to be smooth, with clamped WENO interpolation it even gives visually plausible results at sharp creases.*

A stable interpolation scheme is especially important when dealing with non-smooth surfaces. While the fundamental assumptions of the CPM are true only for smooth surfaces like the analytic sphere shown in Figure 6 middle, it is desirable to also handle triangulated surfaces as well as surfaces with sharp features as shown, for instance, in Figure 6 bottom. However, the closest point field on the concave side of the surface can contain discontinuities even in the vicinity of such features, and due to the closest point extension this can result in discontinuities in the simulation attributes (see Figure 7). This effect is proportional to the dis-

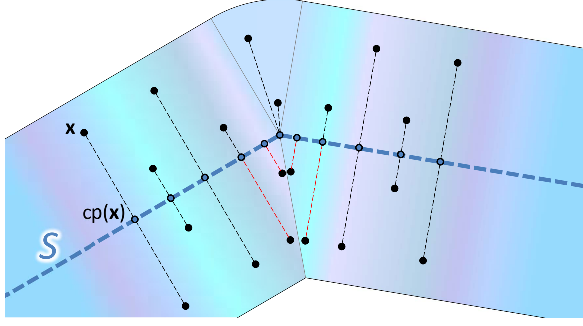

Figure 7: *At sharp creases the closest point field is discontinuous on the concave side of the surface. This can also lead to discontinuities in the simulation attributes.*

tance of the closest points on the two sides of the discontinuity, which itself depends on the sharpness of the feature and the distance from the surface. As a consequence, especially when larger stencils are used the stability of the interpolation scheme becomes important. In general, the same applies to the integration technique used in the evolution phase. Since the proposed integration schemes did not show any stability issues at sharp features, however, special treatment was only necessary in the extension phase.

Linear interpolation is computationally cheap and fulfills the convex hull property, which guarantees stability of the interpolation even if the field is discontinuous. Unfortunately, at least for second-order PDE problems, linear interpolation is not accurate enough for the CPM to ensure consistency [RM08]. At any rate and in practice, linear interpolation introduces a significant amount of artificial numerical diffusion, which manifests in a loss of energy in physics simulations (cf. Figure 6 top).

In the work of Macdonald and Ruuth [MR08], a weighted, essentially non-oscillatory (WENO) interpolation appropriate for the CPM was presented (Section 3.1 and Appendix A of [MR08]). In smooth areas it is of higher order, which prevents numerical diffusion. As with most polynomial interpolations, stability can be critical near discontinuities where the polynomials overshoot the values in the stencil. To reduce this effect, the WENO algorithm calculates a weight for several candidate polynomial interpolants and takes a weighted sum of each interpolant. If a particular candidate is highly oscillatory in a given region it is assigned a very small weight. By this means oscillations are minimized in those areas, as is the formal order-of-accuracy. In the unlikely case that all polynomials oscillate, the WENO interpolation can still overshoot. In our tests, we experienced this effect only for the example shown in Figure 6 bottom, near the inner corners. To ensure stability in this event, we restrict (clamp) the interpolation result to the $[u_{smin}, u_{smax}]$ range, with u_{smin} and u_{smax} being the minimum and maximum values in the interpolation stencil.

The desired interpolation order dictates the size of the stencil and thereby the bandwidth requirements. We use the WENO4 interpolation scheme which is built on quadratic candidate interpolants and recovers tri-cubic interpolation in smooth regions (tri-quadratic otherwise) using 64 entries in the stencil. It is sufficiently accurate for the CPM and we find it to be fast enough for an interactive application.

5.4. Wave equation

The wave equation is the classical example of a hyperbolic equation:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla_S^2 u$$

In our case u represents the height of an elastic surface over time t for a given wave speed c. Note that this is the surface-PDE version of the equation, which means that we can interpret u as the height above some original surface S. Because the right-hand side involves only the Laplace-Beltrami operator, we can simply replace it with the Cartesian Laplace operator to retrieve the embedding PDE for the CPM. We then obtain a discretization for a 3D Cartesian grid with spacing $\Delta\mathbf{x}$ and regular time intervals of length Δt by substituting the second order derivatives on both sides of the equation with finite differences. For each grid point (i, j, k) this leads to:

$$\frac{\tilde{u}^{t+1}_{i,j,k} - 2u^t_{i,j,k} + u^{t-1}_{i,j,k}}{\Delta t^2} = \frac{c^2}{\Delta\mathbf{x}^2}\left(u^t_{i+1,j,k} + u^t_{i-1,j,k}\right.$$
$$\left. + u^t_{i,j+1,k} + u^t_{i,j-1,k} + u^t_{i,j,k+1} + u^t_{i,j,k-1} - 6u^t_{i,j,k}\right)$$

Solving for the unknown $\tilde{u}^{t+1}_{i,j,k}$ leads to the explicit solution

$$\tilde{u}^{t+1}_{i,j,k} = \alpha\cdot\left(u^t_{i+1,j,k} + u^t_{i-1,j,k} + u^t_{i,j+1,k} + u^t_{i,j-1,k}\right.$$
$$\left. + u^t_{i,j,k+1} + u^t_{i,j,k-1}\right) + (2-6\alpha)\cdot u^t_{i,j,k} - u^{t-1}_{i,j,k}$$
$$\text{with } \alpha = \frac{\Delta t^2 \cdot c^2}{\Delta\mathbf{x}^2},$$

followed by a closest point extension

$$u^{t+1}_{i,j,k} := \tilde{u}^{t+1}(cp(\mathbf{x}_{i,j,k})).$$

Note that this is essentially a Verlet integration [Ver67] since the second derivatives in space as well as in time are approximated with central differences. Compared to an explicit Euler scheme, this method improves the accuracy of the integrator to 2$^{\text{nd}}$ order. Since the Verlet integration is stable when the CFL condition is met, i.e., for $\alpha \leq 1/3$ in our case, we always choose this value in all of our experiments. To maximize the accuracy of the simulation, we set the resolution of the computational grid to the highest value which still allows interactive frame rates. Since the integrator requires the height fields of the last two time steps, we copy u^t to u^{t-1} whenever we specify initial conditions or modify the height field during the simulation.

5.5. Incompressible Navier–Stokes equations

An incompressible, Newtonian fluid with constant temperature can be described by two continuity equations: One for momentum

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla_S)\,\mathbf{u} - \frac{1}{\rho}\nabla_S\, p + \nu\nabla_S^2\,\mathbf{u} + \mathbf{f}$$

and one for volume and mass

$$\nabla_S \cdot \mathbf{u} = 0.$$

In this variant of the Navier–Stokes equations—formulated as surface PDE—$\mathbf{u}$ denotes the fluid's velocity, ρ the constant density, p the pressure and ν the constant kinematic viscosity. Consequentially, the terms in the first equation represent accelerations resulting from self-advection $(-(\mathbf{u}\cdot\nabla_S)\,\mathbf{u})$, pressure $\left(-\frac{1}{\rho}\nabla_S\, p\right)$, viscosity $\left(\nu\nabla_S^2\,\mathbf{u}\right)$ and external forces $(\mathbf{f})$, while the second equation states that the velocity field must be free of divergence. Instead of interpreting this flow problem as conservation laws with constraints, we split up the equations and treat each term individually.

The last two terms of the momentum equation are unrepresented in our solver, since highly viscous flows and external forces are currently not considered. If low order interpolation is used in the extension phase, however, the flow already exhibits some artificial viscosity due to numerical diffusion. We therefore also disregard fluids which are completely inviscid. On the other hand, viscous flows can be integrated either explicitly (with a potentially restricted timestep) or using an implicit CPM [MR09], and external forces can be added in a straightforward way.

For the nonlinear convective acceleration $-(\mathbf{u}\cdot\nabla_S)\,\mathbf{u}$, we use a semi-Lagrange back-trace as, e.g., described by Stam [Sta99]. By this means a computationally more expensive implicit solution can be avoided. To update the value of $\mathbf{u}$ at each computational grid vertex, the back-trace follows the motion of a particle starting at the vertex backwards in time through the velocity field to find the previous location of the particle. At this position, the old velocity field is sampled and the retrieved value replaces $\mathbf{u}$ at the grid vertex.

This back-trace could naïvely be implemented in a full simulation grid using its 3D equivalent. In this case, only one change is necessary in order to discretize the embedding of the surface-advection with the back-trace: as the velocity vector must stay in the tangent space also at its new position, it must be projected onto the local tangent plane.

For the sparse grid discussed here, however, it must be ensured that such a back-trace does not leave the computational band. This results in additional constraints on the maximal velocity, the integration time step, and the minimal size of the extension band. To minimize these limitations we perform the back-trace only for points on the surface. Figure 8 illustrates this method.

The back-trace starts at the closest point of the current grid vertex $cp(\mathbf{x}_G)$ and uses a single step of the explicit Euler method. Because the current velocity values are a closest point extension, they are constant in the normal direction and for the start of the back-trace we can directly use the velocity value stored at the grid vertex $\mathbf{x}_G$. At the old particle location $\mathbf{x}^{t-1}$ we sample the velocity field using an interpolation. To avoid numerical diffusion due to the interpolation in the back-trace, we use an interpolation of higher order like the one we described in the context of the closest point extension. The interpolated velocity vector from the particle's old position is projected onto the tangent plane at the new position and then rescaled to its old length in order to retrieve the updated value for the grid vertex.

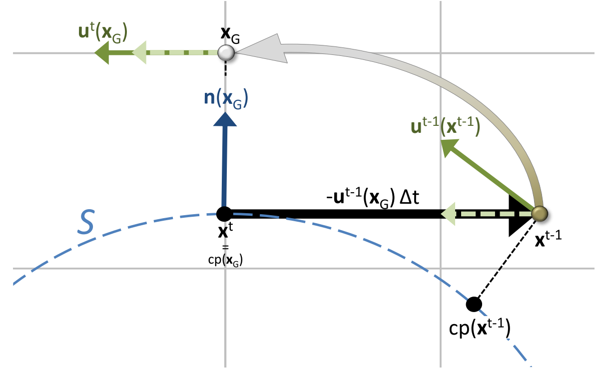

Figure 8: *Velocity self-advection through a semi-Lagrange back-trace in the simulation grid.*

To obtain the surface normal necessary for such a projection at a grid point $\mathbf{x}_G$, one can normalize the difference vector $\mathbf{d} = \mathbf{x}_G - cp(\mathbf{x}_G)$ from the closest surface point to the point itself. These normals always point away from the surface on both of its sides. For non-orientable surfaces, for which both directions are equally valid, this behavior is thoroughly intended. A real problem with this approach, however, is that it fails for grid points very close to or on the surface. To reliably obtain a normal as well for these points, we therefore examine the grid points $\mathbf{x}_i$, $i = 0, 1, \ldots$ in a small stencil around $\mathbf{x}_G$ and calculate a vector $\mathbf{d}_i = \mathbf{x}_i - cp(\mathbf{x}_i)$ for each of them. We accumulate the differences in the vector $\mathbf{d}_S^n$ as follows:

$$\begin{aligned} \mathbf{d}_S^0 &= \mathbf{d}_0 \\ \mathbf{d}_S^{i+1} &= \mathbf{d}_S^i + \begin{cases} \mathbf{d}_{i+1} & \text{if } \mathbf{d}_{i+1}\cdot\mathbf{d}_S^i \geq 0 \\ -\mathbf{d}_{i+1} & \text{otherwise} \end{cases} \end{aligned}$$

The negation is necessary to account for grid points on different sides of the surface. A surface normal is then retrieved by normalization of $\mathbf{d}_S$. Note that the resulting normal points to the side of the surface on which the first off-surface vertex in the stencil lies. In practice, we find using the 6 edge adjacent points around $\mathbf{x}_G$ works well.

In fluid mechanics, the pressure term $-\frac{1}{\rho}\nabla_S\, p$ is often used within a projection method [CM00] to ensure that the resulting velocity field fulfills the incompressibility equation $\nabla_S \cdot \mathbf{u} = 0$. Conceptually, this is achieved by applying a Hodge decomposition to the velocity field, which splits it into a divergence-free field and a curl-free field.

We determine a p that lets $\frac{1}{\rho}\nabla_S\, p$ reproduce the curl-free part, such that after the subtraction only the divergence-free part is left. At every time step, this ultimately requires calculating the divergence of the current velocity and solving a surface Poisson problem of the form $\nabla_S \cdot \nabla_S p = \nabla_S \cdot \mathbf{u}$ to retrieve the desired pressure. Treating this Poisson equation with an artificial time iteration and the explicit CPM would require a closest point extension after each step, which makes this approach quite costly. Another option would be applying the implicit CPM [MR09], either to an artificial time iteration or directly to the Poisson problem. This would require solving a large linear system in each time step.

We choose a computationally cheaper approach where we apply the closest point extension only to the right-hand side of the equation, and we replace the left-hand side with the simpler Cartesian Laplacian. Then, we solve the linear system with the conjugate gradients method [KW03]. The solver considers only grid points within the evolution band and uses a Neumann boundary condition for the values outside (*i.e.,*the gradient in normal direction to the boundary is zero). We chose a Neumann condition because it resembles the effect of the closest point extension within an artificial CPM time-integration [Gre06]. While this may not enforce incompressibility to a high order of accuracy, in practice it gives very good results if we reuse the pressure from the last time step as the initial guess. After the pressure update, a closest point extension must also be applied to the velocity field itself to prepare it for the next time step. It is worth noting here, that the Neumann boundary condition is only necessary to resemble the behavior of the CPM at the boundary of the evaluation band. It may not be confused with the typical CFD boundary conditions used to simulate object boundaries or in-/out-flow conditions.

In order to visualize the fluid flow we introduce an additional, final sub-step in which the advection of a mass-less, colored dye through the velocity field is simulated. The dye is transported through the flow by the same advection operator $\mathbf{u} \cdot \nabla_S$ that is used for the velocity. The only difference is that the dye is represented by three scalar fields—one for each *rgb* color channel—as such the operator does not include a projection onto the tangent plane.

6. Rendering

For rendering the simulated fluid effects on the surface we use Direct3D 10, and we make use of CUDA's interoperability functions to access the respective GPU memory resources within both APIs. If the surface is given as a triangle mesh and the simulated attributes are used to modulate its color, the mesh is rendered via polygon rasterization. The coordinates of the triangle vertices in the embedding 3D domain are interpolated by the rasterizer and then a pixel shader is employed, to retrieve values from the simulation buffers using trilinear interpolation. Figure 9 (left) demonstrates such a rendering with flat shading to show the triangulation, and in Figure 1 (top) Phong shading was used to realize a smooth look of the surface.

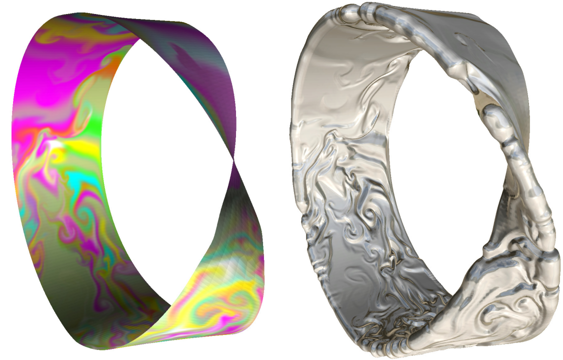

Figure 9: *Navier-Stokes simulation on a Möbius strip; a non-orientable manifold. Left: Rendering with rasterization and flat shading to emphasize the triangle mesh serving as the original surface description. Right: Same simulation as left, but now rendered via raycasting and using the red ink as a displacement on "both sides" of the surface.*

6.1. Volume Raycasting

If the initial surface is not given as a polygon mesh, for instance if it is given by a point set representation or an implicit surface description, or if the simulated attributes should be used to modulate the geometry of the surface, rasterization cannot be used any more. To overcome this problem, we present a GPU method that renders the surface directly from the sparse closest point volume representation, regardless of the initial surface representation.

A first approach is to perform exact voxel ray-casting of the closest point volume in a DDA-like fashion [AW87]. The voxels correspond to the closest point cells, and the constant voxel attributes are given by the simulation values at the cell vertices. The rays of sight first traverse the grid of blocks until a marked block is hit. Then, the block's location in the *closest-point-buffer* is retrieved and traversal is continued on the cells inside. Traversal is stopped if a voxel is hit that contains the closest surface point stored at this voxel, and the voxel attribute is used as pixel color.

While the proposed technique can visualize a piecewise (per voxel) constant distribution of attributes across a surface, it does not allow rendering a smooth distribution. This is achieved by interpolating the distances of the cell vertices

to their closest surface points and performing an exact intersection test between the ray and its zero crossings in the reconstructed field.

For this purpose we sample along the ray in regular intervals of half the voxel width. When a hit is detected, we refine the result using binary search. Since the CPM is based solely on the existence of an *unsigned* distance transform, the zero crossings cannot be calculated exactly by this means, and the intersection test comes down to finding the points along the rays where the distance between the sample position and the trilinearly interpolated closest point is below a given ε. At this intersection point, the simulation attributes are retrieved via trilinear interpolation, too. To simulate smooth shading, the attributes are translated into a material color and per-pixel normals are calculated by normalizing the vector between the intersection point and the closest surface point.

6.2. Rendering Surface Displacements

In addition to using the closest point representation for rendering the original surface, we can also use it to for simulating surface displacements that are given, for instance, by the simulation values at the grid vertices (see Figures 1 bottom, and 9 right). After interpolating the closest point $cp(\mathbf{x}_R)$ at a sample position $\mathbf{x}_R$ on a ray, the simulation value $h(cp(\mathbf{x}_R))$ at this point is interpolated and compared to the distance between $cp(\mathbf{x}_R)$ and $\mathbf{x}_R$. As Figure 10 shows, a hit is detected if the distance value is smaller than the interpolated attribute value. The figure also shows, that the interpolation of closest points from different parts of the surface can lead to off-surface points. To avoid rendering artifacts in this case, we determine the maximum extent of an axis aligned bounding box around the closest points of the interpolation stencil for the current ray sample. If this bounding box is considerably larger than the interpolation cell, we skip the current sample and continue sampling along the ray. The normal of the displaced surface is calculated from the original surface normal and the gradient of the attribute field. Since the gradient is tangent to the surface because of the CPM, it can be used in turn to disturb the surface normal appropriately. The maximum displacement is supposed not to exceed the width of the computational band. This constraint can be abandoned, however, by using an enlarged displacement band, where the grid vertices outside the computational band store only their closest points but no attributes.

The proposed rendering approach displaces the surface equally on both of its sides. For orientable surfaces it is also possible to distinguish the two half-spaces in order to displace in just one direction or to use negative displacements. For non-orientable surfaces the view-direction can be used to introduce an artificial orientation. Despite being view dependent, this approach has the additional problem that it results in artifacts at the silhouettes, where the orientation flips. We therefore recommend to treat both sides equally in this case.

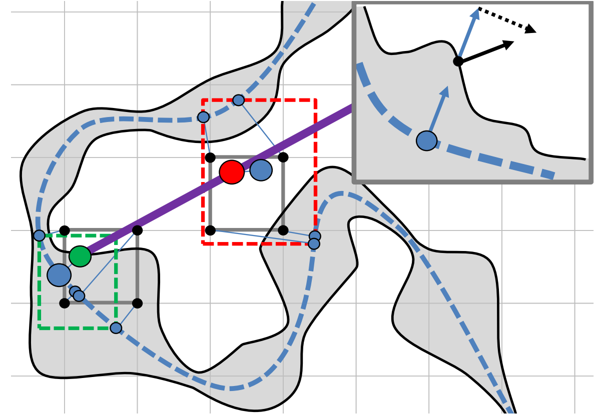

Figure 10: *Bottom left: Intersection of a ray (purple) with a displaced surface (black). At the red ray sample, the interpolation of closest points on the original surface (blue) results in an off-surface point. At the green ray sample, an intersection with the displaced surface was found, as the distance to the closest point equals the interpolated height value. Top right: The normal of the original surface (blue arrows) combined with the gradient of the height value (dotted arrow) leads to the normal of the displaced surface (black arrow).*

7. Performance Analysis

To validate the efficiency of the proposed method for simulating and rendering fluid effects on surfaces, we have performed a number of experiments using surface models and simulation grids at different resolutions. In all of our experiments, a block size of $b = 4$ was used. We found this size to give the best performance compared to smaller (tighter fit of the adaptive simulation grid to the surface but increasing number of indirections to access adjacent grid cells) and larger (increasing memory and computation requirements) blocks. All measurements were performed on a 2.4 GHz Core 2 Duo processor and an NVIDIA GeForce GTX 480 graphics card with 1,536 MiB local video memory. A detailed memory and performance statistic of the GPU CPM for fluid simulation is given in Table 1. Here, numbers separated by a slash refer to the simulation using the wave equation with linear interpolation and the Navier-Stokes equations with WENO4 interpolation, respectively. All timings are given in milliseconds and rendering was always performed on a $1,024 \times 1,024$ viewport.

For each model, the first column lists the name and number of triangles, while the second column gives the resolution of the uniform Cartesian grid to which the simulation resolution corresponds. The third column lists the number of closest points within the computational band. The differences are due to differently sized computational bands that are dictated by the respective numerical stencils of the wave equation (bandwidth with linear interpolation: $2.4\Delta\mathbf{x}$) and the Navier-Stokes equations (bandwidth with WENO4

interpolation: $5.7\Delta\mathbf{x}$). The forth column gives the memory that is required for solving the equations on the GPU via the CPM. The higher memory consumption of the Navier-Stokes simulation is due to the larger computational band and the additional buffers that are required to store the simulation attributes. The times required to build the closest point representation are given in the fifth column. Here it is important to note that more than 75% of the time is always required by the closest points computation, meaning that the grid construction on the GPU consumes only a small portion of this time. It can further be seen that the GPU memory requirements mainly depend on the resolution of the simulation grid, while on the other hand, the time for constructing the grid is strongly dependent on the number of triangles.

The following four columns list the simulation times using linear and WENO4 interpolation in the closest point extension for the wave equation and the Navier-Stokes equations. As expected, WENO4 interpolation increases the overall simulation times significantly due to its larger computational stencil, and the enlarged computational band thereof. The Navier-Stokes simulation uses the respective interpolation type in three separate closest point extensions (divergence, velocity, ink) and two advection passes (velocity, ink). It can be observed that solving the pressure Poisson equation is the most expensive operation (~75% of the total time) as long as linear interpolation is used. With WENO4 interpolation, however, these ratios turn into the opposite: Most of the time is now spent on interpolations in the advection (~50%) and extension kernels (~40%).

The last three columns show the rendering times for rasterization, voxel-based surface raycasting, and raycasting with surface displacements. The timings refer to the rendering of a WENO4 wave simulation. In the last column a computational band of width $8\Delta\mathbf{x}$ was used to allow simulating large surface displacements. As expected, rasterizing the triangle mesh on the GPU leads to the highest frame rates. Direct surface raycasting is between 1.2 to 5 times slower, but still delivers highly interactive frame rates. Raycasting with smooth displacements, which allows simulating dynamic surface modifications, also delivers interactive frame rates even for the computational grids with the highest resolution. This is quite remarkable since much larger closest point and simulation buffers are used.

7.1. Surface Deformations

The timing statistics indicate that the creation of an adaptive closest point grid for the CPM is possible at high rates even for large triangular meshes. This is an important step towards a real-time CPM for simulating deformations of the surface itself. In the following we demonstrate mesh smoothing based on the Laplace operator [Tau95] via the GPU CPM as a first example (see Fig. 11).

Each smoothing step consists of three operations:

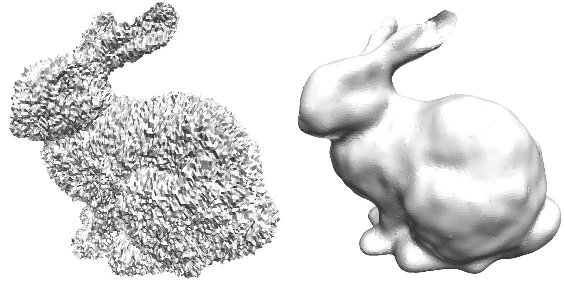

Figure 11: *Laplacian mesh smoothing using the CPM. Left: Distorted Bunny model. Right: The model after 15 smoothing steps (64ms each) using a grid resolution of* 256^3. *Both images show triangle rasterization and flat shading.*

Firstly, we create a discrete, adaptive closest point field as described. Secondly, we apply the Laplace-Beltrami operator—discretized with the standard Laplace operator on the embedding Cartesian grid—to the closest point field. At every point on the surface the resulting vector field is directed towards the mean of the point's neighborhood on the surface. Thirdly, we interpolate the discrete field using WENO4 at every vertex position of the mesh and move each vertex into this direction.

8. Conclusion and Future Work

We have presented a real-time method for simulating fluid effects on surfaces using the CPM. To achieve this, we have developed a fast GPU method for realizing the CPM and rendering the simulation results as surface colors or displacements. The method can efficiently solve non-linear PDEs on surfaces using a high-resolution embedding grid, and it can calculate and render the resulting fields independently of the resolution and topology of the input geometry.

In the presented examples the CPM is solely used for solving PDEs on (infinitely thin) surfaces, without enforcing any problem-specific physical conditions. Even though this would be possible in special cases, it has not been considered in this work. Consequently, the effects demonstrated in this work do not show real fluid effects as they would appear in reality, but rather demonstrate the potential of the CPM for producing visually plausible and compelling fluid effects on surfaces in real-time. It is interesting to note, on the other hand, that the exact same type of PDEs may arise at the boundaries of volumetric simulations, and coupling oddly-shaped boundaries to full volumetric simulations opens an interesting venue for future work.

The proposed methods open a number of future research directions. In particular, we would like to adopt the CPM for real-time effects on animated surfaces. So far, we are able to construct an adaptive embedding grid for the deforming surface in every frame. Further research is necessary, however, to define a CPM for PDEs on deforming surfaces and

Table 1: *Performance statistics for fluid simulation and rendering on the GPU via the CPM.*

		Grid Generation			Wave Equation		Navier-Stokes		Rendering		
Mesh	Res.	# Closest Points	GPU Memory	Time	linear	WENO4	linear	WENO4	Rasterization	Raycasting	Displ.
Möbius	64^3	34k / 48k	1MiB / 4MiB	6.1ms / 9.5ms	2.0ms	2.1ms	3.7ms	20ms	1.3ms	3.8ms	20ms
(Figure 9)	128^3	77k / 184k	4MiB / 18MiB	6.4ms / 10ms	2.1ms	4.1ms	7.5ms	65ms	1.3ms	4.3ms	23ms
3,385Δ	256^3	303k / 721k	18MiB / 73MiB	7.4ms / 12ms	2.3ms	15ms	23ms	237ms	1.4ms	5.5ms	31ms
	512^3	1.2M / 2.9M	82MiB / 297MiB	12ms / 21ms	5.1ms	54ms	85ms	909ms	1.7ms	8.4ms	48ms
Bunny	64^3	41k / 80k	2MiB / 7MiB	29ms / 84ms	2.2ms	3.0ms	4.0ms	28ms	1.4ms	3.3ms	15ms
(Figure 1)	128^3	177k / 374k	9MiB / 35MiB	32ms / 86ms	2.4ms	9.0ms	13ms	118ms	1.4ms	4.0ms	24ms
69,451Δ	256^3	733k / 1.6M	41MiB / 159MiB	34ms / 90ms	3.6ms	34ms	51ms	529ms	1.4ms	5.7ms	35ms
	512^3	3.0M / 6.8M	178MiB / 679MiB	49ms / 117ms	11ms	136ms	222ms	2,174ms	1.7ms	8.6ms	52ms
Armadillo	64^3	28k / 58k	1MiB / 5MiB	117ms / 377ms	2.0ms	2.7ms	3.4ms	18ms	2.5ms	3.0ms	16ms
(Figure 1)	128^3	123k / 258k	6MiB / 25MiB	118ms / 380ms	2.1ms	6.1ms	9.5ms	80ms	2.5ms	3.7ms	20ms
345,944Δ	256^3	513k / 1.1M	29MiB / 110MiB	120ms / 384ms	2.9ms	25ms	38ms	370ms	2.5ms	4.6ms	28ms
	512^3	2.0M / 4.8M	129MiB / 477MiB	135ms / 404ms	8.6ms	100ms	157ms	1,667ms	2.6ms	5.9ms	45ms

to develop an efficient solution for the frame-to-frame information flow. Due to the independence of the CPM and our rendering technique from the original surface representation, such a method would also allow simulating complex deformations of the surface itself.

From a numerical point of view we are most interested in exploring how multigrid schemes can be employed to speed up the simulation on the embedding grid. This requires pursuing research on the construction of a multigrid closest point representation that can accurately approximate the surface at ever coarser scales. By integrating such a construction into approaches that can create a coarse grid hierarchy independently of the complexity of the object's shape [LPR*09, DGW11], good approximation quality and convergence rates can be expected even for complicated surfaces.

Acknowledgements

This publication was based on work supported in part by the Munich Centre of Advanced Computing at the Technische Universität München (TUM) and by Awards No. KUK-C1-013-04 and UK-C0020, made by King Abdullah University of Science and Technology (KAUST).

References

[ATBG08] ANGST R., THÜREY N., BOTSCH M., GROSS M.: Robust and efficient wave simulations on deforming meshes. *Computer Graphics Forum (Proc. Pacific Graphics) 27* (2008), 1895–1900. 3

[AW87] AMANATIDES J., WOO A.: A fast voxel traversal algorithm for ray tracing. In *Eurographics '87* (1987), pp. 3–10. 10

[BCOS01] BERTALMÍO M., CHENG L.-T., OSHER S., SAPIRO G.: Variational problems and partial differential equations on implicit surfaces. *J. Comput. Physics 174*, 2 (2001), 759–780. 1

[BKW10] BÜRGER K., KRÜGER J., WESTERMANN R.: Sample-based surface coloring. *IEEE Transactions on Visualization and Computer Graphics 16*, 5 (2010), 763–776. 3

[CLB*09] CHUANG M., LUO L., BROWN B. J., RUSINKIEWICZ S., KAZHDAN M.: Estimating the Laplace-Beltrami operator by restricting 3D functions. In *Proceedings of the Eurographics Symposium on Geometry Processing* (2009), pp. 1475–1484. 1, 3

[CM00] CHORIN A., MARSDEN J.: *A Mathematical Introduction to Fluid Mechanics*, 4th ed. Springer Verlag, 2000, ch. 1.3, pp. 36–44. 10

[CRT04] CLARENZ U., RUMPF M., TELEA A.: Surface processing methods for point sets using finite elements. *Computers and Graphics 28*, 6 (2004), 851–868. 1

[DCB*04] DONG Z., CHEN W., BAO H., ZHANG H., PENG Q.: Real-time voxelization for complex polygonal models. In *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference* (2004), PG '04, pp. 43–50. 3

[DE07] DZIUK G., ELLIOTT C.: Surface finite elements for parabolic equations. *J. Comput. Math. 25* (2007), 385–407. 1

[DGW11] DICK C., GEORGII J., WESTERMANN R.: A hexahedral multigrid approach for simulating cuts in deformable objects. *IEEE Transactions on Visualization and Computer Graphics 17* (2011), 1663–1675. 13

[ED06] EISEMANN E., DÉCORET X.: Fast scene voxelization and applications. In *ACM SIGGRAPH 2006 Sketches* (2006). 3

[FZKH05] FAN Z., ZHAO Y., KAUFMAN A., HE Y.: Adapted unstructured LBM for flow simulation on curved surfaces. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2005), SCA '05, pp. 245–254. 3

[Gre06] GREER J. B.: An improvement of a recent Eulerian method for solving PDEs on general geometries. *J. Sci. Comput. 29* (December 2006), 321–352. 10

[Har07] HARRIS M.: Parallel prefix sum (scan) with CUDA. NVIDIA Whitepaper, April 2007. 5, 6

[HZQW10] HONG Y., ZHU D., QIU X., WANG Z.: Geometry-based control of fire simulation. *The Visual Computer 26* (September 2010), 1217–1228. 3

[JBS06] JONES M. W., BÃĘRENTZEN J. A., SRAMEK M.: 3D distance fields: A survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics 12* (2006), 581–599. 3

[KW03] KRÜGER J., WESTERMANN R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG) 22*, 3 (2003), 908–916. 10

[LFW07] LO K.-Y., FU C.-W., WONG T.-T.: Interactive reaction-diffusion on surface tiles. In *Proceedings of Pacific Graphics* (2007), pp. 65–74. 1

[LPR*09] LIEHR F., PREUSSER T., RUMPF M., SAUTER S., SCHWEN L. O.: Composite finite elements for 3D image based computing. *Comput. Vis. Sci. 12* (March 2009), 171–188. 13

[LWC05] LUI L. M., WANG Y., CHAN T. F.: Solving PDEs on manifolds with global conformal parametrization. In *VLSM* (2005), pp. 307–319. 1

[MR08] MACDONALD C. B., RUUTH S. J.: Level set equations on surfaces via the Closest Point Method. *J. Sci. Comput. 35*, 2–3 (June 2008), 219–240. 3, 4, 8

[MR09] MACDONALD C. B., RUUTH S. J.: The implicit Closest Point Method for the numerical solution of partial differential equations on surfaces. *SIAM J. Sci. Comput. 31*, 6 (2009), 4330–4350. 3, 9, 10

[NMZ07] NEILL P., METOYER R., ZHANG E.: Fluid flow on interacting deformable surfaces. In *ACM SIGGRAPH 2007 posters* (2007), SIGGRAPH '07. 3

[NNRW09] NEMITZ O., NIELSEN M. B., RUMPF M., WHITAKER R.: Finite element methods on very large, dynamic tubular grid encoded implicit surfaces. *SIAM J. on Scientific Computing 31*, 3 (2009), 2258–2281. 1

[Pan11] PANTALEONI J.: VoxelPipe: A programmable pipeline for 3D voxelization. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (2011), HPG '11, pp. 99–106. 3, 4

[RM08] RUUTH S. J., MERRIMAN B.: A simple embedding method for solving partial differential equations on surfaces. *J. Comput. Phys. 227* (January 2008), 1943–1961. 1, 2, 3, 8

[RWP06] REUTER M., WOLTER F.-E., PEINECKE N.: Laplace-Beltrami spectra as "Shape-DNA" of surfaces and solids. *Computer-Aided Design 38*, 4 (2006), 342–366. 3

[SS10] SCHWARZ M., SEIDEL H.-P.: Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia) 29* (2010), 179:1–179:9. 3, 4

[Sta99] STAM J.: Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), SIGGRAPH '99, pp. 121–128. 2, 9

[Sta03] STAM J.: Flows on surfaces of arbitrary topology. *ACM Trans. Graph. 22* (July 2003), 724–731. 1, 2

[SY04] SHI L., YU Y.: Inviscid and incompressible fluid simulation on triangle meshes: Research articles. *Comput. Animat. Virtual Worlds 15* (July 2004), 173–181. 3

[Tau95] TAUBIN G.: A signal processing approach to fair surface design. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995), SIGGRAPH '95, pp. 351–358. 12

[TMR09] TIAN L., MACDONALD C. B., RUUTH S. J.: Segmentation on surfaces with the Closest Point Method. In *Proc. ICIP09, 16th IEEE International Conference on Image Processing* (2009), pp. 3009–3012. 3

[Tur91] TURK G.: Generating textures on arbitrary surfaces using reaction-diffusion. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (1991), SIGGRAPH '91, pp. 289–298. 1, 3

[Ver67] VERLET L.: Computer "experiments" on classical fluids. I. thermodynamical properties of Lennard-Jones molecules. *Phys. Rev. 159*, 1 (Jul 1967), 98. 8

[WMT07] WANG H., MILLER G., TURK G.: Solving general shallow wave equations on surfaces. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2007), SCA '07, pp. 229–238. 3

[ZCEP07] ZHANG L., CHEN W., EBERT D. S., PENG Q.: Conservative voxelization. *Vis. Comput. 23* (August 2007), 783–792. 3