Sparse Surface Speed Evaluation on a Dynamic Three-Dimensional Surface Using an Iterative Partitioning Scheme

Paul Manstetten[1(✉)], Lukas Gnam[1], Andreas Hössinger[2], Siegfried Selberherr[3], and Josef Weinbub[1]

[1] Christian Doppler Laboratory for High Performance TCAD, Institute for Microelectronics, TU Wien, Vienna, Austria
{manstetten,gnam,weinbub}@iue.tuwien.ac.at
[2] Silvaco Europe Ltd., St Ives, UK
andreas.hoessinger@silvaco.com
[3] Institute for Microelectronics, TU Wien, Vienna, Austria
selberherr@iue.tuwien.ac.at

Abstract. We focus on a surface evolution problem where the local surface speed depends on a computationally expensive scalar function with non-local properties. The local surface speed must be re-evaluated in each time step, even for non-moving parts of the surface, due to possibly changed properties in remote regions of the simulation domain. We present a method to evaluate the surface speed only on a sparse set of points to reduce the computational effort. This sparse set of points is generated according to application-specific requirements using an iterative partitioning scheme. We diffuse the result of a constant extrapolation in the neighborhood of the sparse points to obtain an approximation to a linear interpolation between the sparse points.

We demonstrate the method for a surface evolving with a local surface speed depending on the incident flux from a source plane above the surface. The obtained speedups range from 2 to 8 and the surface deviation is less than 3 grid-cells for all evaluated test cases.

Keywords: Surface mesh · Surface evolution · Interpolation
Robust · Scalar · Sparse evaluation

1 Introduction

The simulation of dynamic surfaces is an integral part of a large number of areas including fluid simulations [5], computer graphics [4], and semiconductor fabrication [9]. The maximum time step for the simulation of the dynamic surface is limited by the underlying discretization, the advection scheme, and the maximum surface speed. If the surface speed depends on global properties of the domain, it must be re-evaluated in each time step, even for non-moving parts of the surface. This full re-evaluation – especially for high resolution simulations – potentially leads to situations where the surface speed model evaluation dominates the overall simulation run time.

Y. Shi et al. (Eds.): ICCS 2018, LNCS 10860, pp. 694–707, 2018.
https://doi.org/10.1007/978-3-319-93698-7_53

The approach presented in the following provides a robust method to reduce the number of necessary evaluations of the surface model. From a dense set of evaluation points given by the resolution of the surface mesh, a subset of points is selected using an iterative partitioning scheme. The scheme is controlled by a freely definable refinement condition, allowing to adopt the method for different application-specific requirements. After evaluating the surface model for this subset of points, the solution for the remaining points in the dense set is obtained by diffusing the result of a constant extrapolation in the neighborhood of the sparse points using the error smoothing properties of the Jacobi method [1, p. 895].

We evaluate our method based on an etching simulation problem, taken from the area of semiconductor fabrication. We use a generic etching simulation test case with a single material region to investigate our method. The refinement condition for the iterative partitioning scheme is modeled using fixed thresholds for local flux differences and surface normal deviations. As illustrating example, a study of an etching process involving high aspect ratio holes can be found in [2]. The etching process selectively removes material from a substrate, representing a surface evolution problem. When modeling etching processes, typically the surface speed evaluation is the dominating part of the overall simulation run time. This, together with the fact that simulations become more and more intricate (i.e., both with respect to geometry and involved physics), leads to unacceptable long simulation run times. The central motivation for this work is to reduce the simulation run time in such scenarios as much as possible to enable more intricate simulation problems.

2 Iterative Partitioning Scheme

We require a triangulated surface mesh – representing the evolving surface – and define the *dense* set of evaluation points as the set of all triangle centroids. Algorithm 1 is used to iteratively select a *sparse* subset of evaluation points depending on (a) the maximal globally allowed edge distance (d_{max_0}) between two points in the subset, (b) an array of maximal allowed edge distances for each point in the dense set where each entry is between 0 and d_{max_0}, and (c) a refinement condition. The refinement condition defines in each iteration and for each point in the sparse set, if additional points in the surrounding should be added to the sparse set. Additionally to the sparse set, Algorithm 1 assigns one of the sparse points to each of the points in the dense set. All points with the same sparse "parent" are referred to as *patch* in the following. The patches are the "spacers" between the points in the sparse set and are used to efficiently identify neighbors in the sparse set and to generate the initial guess for the Jacobi solver discussed in Sect. 3. The refinement condition used in Sect. 4 is based on a fixed threshold for the angular deviation of the surface normal and the deviation of the surface speed between a sparse point and its sparse neighbors (cf. Eq. 2).

Details for the subroutines in Algorithm 1 can be found in Appendix A. Figure 1 illustrates the individual stages of the algorithm using a small, regular triangulated mesh:

(a) In the initial iteration, a triangle is selected as active and a patch (red) is formed out of the surrounding triangles until the maximum allowed distance d_{max_0} is reached. In this example, we use $d_{max_0} = 8$. The first triangle is selected arbitrarily; the simplest choice is the first triangle in the list of triangles of the mesh.
(b) One of the remaining triangles (the simplest choice is again the first unprocessed triangle in the list of triangles) in the unprocessed region of the mesh is selected and a new patch (blue) is formed, which overwrites the red patch where the edge distance is smaller to the blue origin. In the initial iteration, this procedure is repeated until all triangles of the mesh have been processed. The result of (b) is a list of patches covering the whole surface.
(c) The connection between the two points of the sparse set (black line) is detected, when the two corresponding patches share one or more edges of the mesh. The result of step (c) is a set of connections between neighboring sparse points (triangles with label "0").
(d) In the next iteration (first refining iteration), if the refinement condition is evaluated to true for a sparse point, the sub region of the associated patch, where the edge distance to the origin is above $d_{max_0}/4$, is withdrawn from the patch; the threshold for the withdrawal ($d_{max_0}/4$) results in a core patch of "diameter" $d_{max_0}/2$, surrounded by a withdrawn region with a minimum "thickness" of $d_{max_0}/4$.
(e) In the withdrawn region, patches are created (analog to the initial iteration) until all triangles have been processed, but now using $d_{max_1} = d_{max_0}/2$; the division by 2 leads to a bisection of the maximal edge distance between sparse points on the patch. Like before, which of the triangles is selected as origin of a patch is arbitrary as long as it is unprocessed; typically the first triangle in the list of withdrawn triangles is chosen.
(f) The connections between the 6 sparse points, as a result of the refinement of the red patch, are illustrated (black lines).

After the refinement is completed for all patches where the refinement condition evaluates to true, the refinement condition is re-evaluated for all sparse points. Subsequently, the refinement is repeated with $d_{max_2} = d_{max_1}/2$, continuously leading to a bisection of the maximal edge distance between sparse points on the patch. The algorithm is terminated either because the refinement condition evaluates to false for all sparse points or $d_{max_i} = 1$, corresponding to a patch consisting of only one triangle. If the refinement condition depends on the surface velocity at the sparse points, the surface model must be evaluated for the newly added sparse points in each iteration.

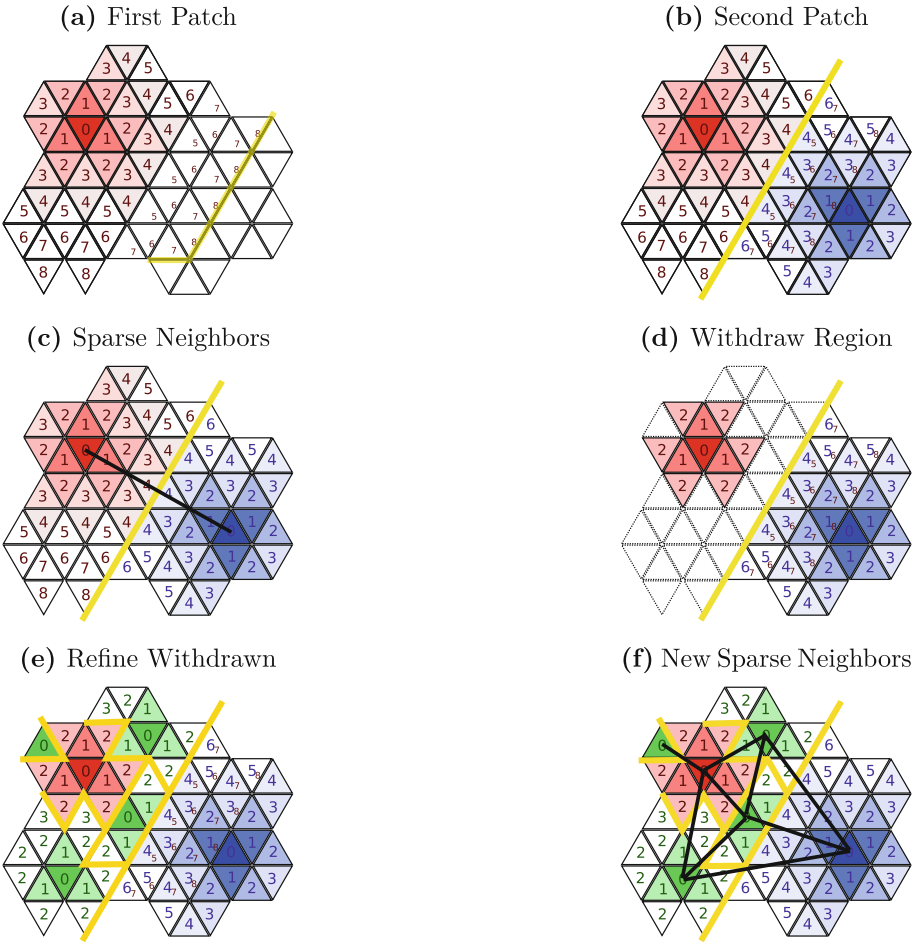


Fig. 1. Schematic depiction of the stages of the iterative partitioning scheme for an exemplary mesh. Yellow lines denote patch boundaries. (a) Initial creation of a patch where the numbers refer to edge distances to the origin; for visualization purposes only, the triangles which will be removed from the patch in the next step use a smaller font size. (b) Creation of a second patch (blue) starting at one of the unprocessed triangles. (c) Sparse neighbor connection between the two origins of the patches. (d) Withdrawal of subregion in red patch. (e) Refinement of the withdrawn region in the red patch with $d_{max_1} = d_{max_0}/2$. (f) Updated sparse neighbor connections after the red patch was refined. (Color figure online)

How well the edge distances between the centroids map to arc length distances on the triangular mesh depends on the uniformity of the mesh with respect to triangle shape and size. Only with a mesh consisting of triangles with comparable size and quality the algorithm will produce “convex” patches (convex with respect to the polygon constructed out of the outermost centroids).

After completion, Algorithm 1 provides a sparse set of points with corresponding sparse neighbors and patch information.

Algorithm 1. Adaptive decimation of evaluation locations on a triangular surface mesh.

Input: d_{max0}, distTarget[i], RefinementCondition(i)
Output: active[], sparseNeighbors[], patches[]
Algorithm

```
withdrawn[N_tri] = true; reflagged[N_tri] = false; active[N_tri] = false;
distance[N_tri] = d_max0; parent[N_tri] = -1;
patches[] = empty map(activeIndex,patchIndices);
sparseNeighbors[] = empty map(activeIndex,activeNeighbors);
indices[N_tri] = iota(0,N_tri);
FlagTriangles(indices, d_max)
RebuildPatches();
EvaluateSurfaceModel(for all active indices)
for n=1...log2(d_max0) do
    reflagged[N_tri] = false;
    withdrawn[N_tri] = false;
    numNewPatches = 0;
    foreach patch in patches do
        i_active = patch.activeIndex;
        if RefinementCondition(i_active) == true AND
        reflagged[i_active] == false then
            numNewPatches += RefinePatch(i_active, d_max0/2^n);
    if numNewPatches == 0 then
        break;
    RebuildPatches();
    EvaluateSurfaceModel(for all newly active indices)
```

3 Interpolation Between Sparse Points

Inherent to its construction method, the sparse set of points and the connections between sparse neighbors do not necessarily allow to construct a sparse mesh covering the complete original surface, which could be used for interpolation. To provide a robust, non-supervised, and computationally efficient interpolation between the sparse points, we start with a constant extrapolation inside the patches using the corresponding values at the origins. We use the properties of Laplace's equation (Eq. 1) and the error diffusion properties of the Jacobi method to smooth the jumps in the constant extrapolation and to approximate a linear interpolation between the sparse points:
(a) In one dimension, the solution of Laplace's equation (Eq. 1) is equivalent to a linear interpolation between a sparse set of points when using the sparse set as Dirichlet boundary conditions and model the boundaries of the domain as zero gradient Neumann boundary conditions.

(b) In one iteration of Jacobi's method, local information travels only across one edge; using this property we can restrict the radius of influence to not exceed the maximal patch radius of $d_{max_0}/2$ by only performing $d_{max_0}/2$ or less iterations.

We approximate a linear interpolation between the sparse points on the surface by using the same boundary conditions (cf. (a)) and starting with the constant extrapolation as an initial guess. We do not solve Eq. 1 until convergence but only perform a fixed number of iterations of Jacobi's method.

$$-\nabla^2 \mathbf{u} = 0 \tag{1}$$

We use a finite volume approximation to discretize Eq. 1 on the triangulated mesh by

integrating over the volume $$-\int_{V_i} \nabla^2 \mathbf{u} \, \mathrm{d}V = 0,$$

applying Green's Theorem $$-\int_{\delta V_i} \nabla \mathbf{u} \cdot \mathbf{n_i} \, \mathrm{d}S = 0,$$

summing over the triangle edges $$-\sum_{j=1}^{3} \int_{\delta V_{ij}} \nabla \mathbf{u} \cdot \mathbf{n_{ij}} = 0,$$

using the midpoint rule $$-\sum_{j=1}^{3} L_{E_{ij}} \nabla \mathbf{u} \cdot \mathbf{n_{ij}} \, \mathrm{d}S = 0, \quad \text{and}$$

using a central difference between centroids $$\nabla \mathbf{u} \cdot \mathbf{n_{ij}} \approx \frac{\mathbf{u}(x_{n_{ij}}) - \mathbf{u}(x_i)}{\| \, x_{n_{ij}} - x_i \, \|},$$

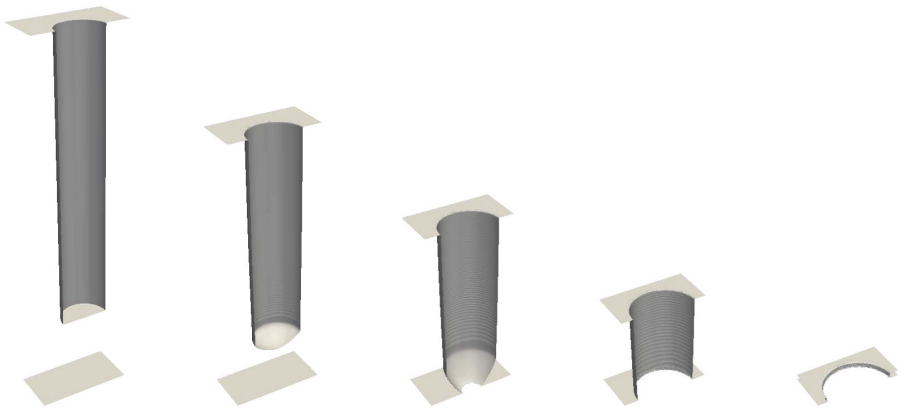
where $\mathbf{u}$ is the scalar function (in this case the local surface velocity), $L_{E_{ij}}$ is the length of the edge shared by triangle i and j, and x_i is the centroid of triangle i, and $x_{n_{ij}}$ is the centroid of the triangle connected to triangle i across edge j. Using this discretization and the boundary conditions described above results in a system of linear equations. The number of unknowns is the number of all centroids minus the size of the sparse set.

4 Results

We evaluate our method using a generic etching simulation test case with a single material region. The model for the surface speed is a linear relation to the direct incident flux from a remote source plane above the surface. All results were produced using a vertically focused ($n = 100$) power cosine source distribution $\Gamma(\Theta) = cos(\Theta)^n$. We use an integration method based on a 5 times subdivided icosahedron as presented in [6] to calculate the direct flux rates on the surface. The direct flux rates are normalized to the flux rate on a fully exposed horizontal plane. As in [6], we used *Embree* [3,10] as ray tracing engine and *OpenVDB* [7,8] for level-set based surface advection and extraction.

The initial geometry (Fig. 2a) is a cylindrical hole with diameter 1 and depth 6 in a bulk region of thickness 8. Figure 2b–e show the intermediate surface

positions using the dense centroid-set for surface model evaluation (dense flux evaluation) from time T = 0 up to T = 8, where the bulk region is completely etched.



(a) T = 0 (b) T = 2 (c) T = 4 (d) T = 6 (e) T = 8

Fig. 2. Cylindrical hole with diameter 1 and depth 6 in a bulk region of thickness 8. Surface evolution during the simulation at times T = [0, 2, 4, 6, 8] (all units are arbitrary). The level-set resolution is 64 cells per unit length.

To model the refinement condition we define for each sparse point i,

the maximal normal deviation $$\nu_{max_i} = \max_{\forall k \in N_k} \angle(\mathbf{n}_i, \mathbf{n}_k),$$

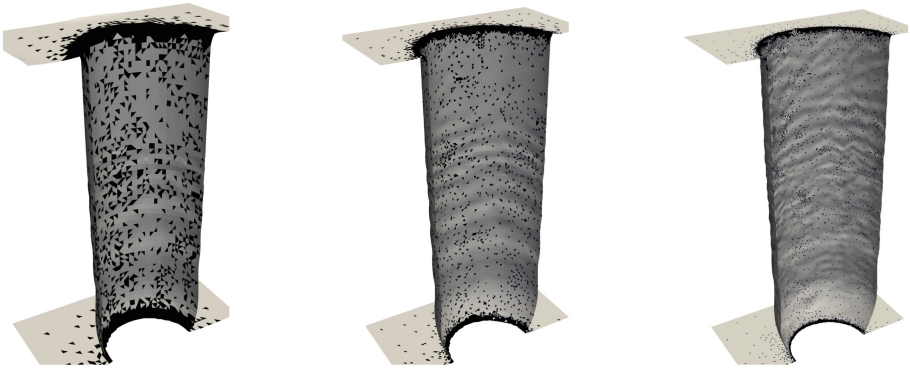
the average flux difference $$du_{avg_i} = \frac{1}{n(N_k)} \sum_{\forall k \in N_k} \frac{|u_i - u_k|}{|u_{max} - u_{min}|},$$

and the maximal flux difference $$du_{max_i} = \max_{\forall k \in N_k} \frac{|u_i - u_k|}{|u_{max} - u_{min}|},$$

where N_k is the set of neighboring sparse point indices, and u_{max} and u_{min} are the global maximum and minimum flux value, respectively. We use a combination of fixed thresholds in all of the following results to model the refinement condition in Algorithm 1:

$$RefinementCondition(i) = \begin{cases} true, & \text{if } \nu_{max_i} > \pi/10 \\ true, & \text{if } \frac{du_{avg_i} + du_{max_i}}{2} > 0.2 \\ false, & \text{otherwise} \end{cases} \tag{2}$$

Furthermore, we use $d_{max_0} = 32$ in all simulations, which gives a total of 6 iterations (1 initial iteration, and $log2(d_{max_0}) = 5$ refinements), whereas the number of Jacobi iterations is fixed to $d_{max_0}/4 = 8$. Figure 3 illustrates the resulting sparse centroid-set at time T = 4.5 for different level-set resolutions.



(a) 15412/3264 ≈ 5 **(b)** 56680/5802 ≈ 10 **(c)** 220580/12611 ≈ 17

Fig. 3. Sparse set of triangles (cf. triangles with labeled with "0" in Fig. 1) for level-set resolutions 16 (a), 32 (b), and 64 (c) at time T = 4.5. The ratios between the total number of triangles and the sparse set of triangles (black) are denoted.

In Fig. 4, we compare the results between the dense and sparse flux evaluation at times T = 3 and T = 6. For a level-set resolution 64, this corresponds to time step 800 and 1600, respectively. The two surfaces reveal deviations of up to 3 level-set cells, most prominent in the lower vertical region of the hole. In the upper region of the hole and the top surface, nearly no deviations are present.

Table 1. Level-set resolutions, resulting initial domain resolutions, initial mesh properties, and resulting number of time steps until T = 8.

Cells per unit length	Cells vertical	Cells horizontal	Triangles	Time steps
16	128	32×32	17k	540
32	256	64×64	67k	1080
64	512	128×128	262k	2160

We evaluate the performance of our method by tracing the run time per time step from T = 0 to T = 8 for three different level-set resolutions summarized in Table 1 for the dense and the sparse flux calculation. For each time step, the run time for the flux evaluation and for the remaining parts (velocity extension, advection, normalization, and mesh extraction; referred to as *other tasks* in the following) is tracked (cf. Figs. 5, 6 and 7 green and red areas, respectively). The flux integration method, which is used for a single point, is identical for both cases. The implementation of Algorithm 1 is serial, in contrast to the flux evaluation, which is OpenMP-parallelized in both cases to form a basis for a realistic estimation of the speedups. The serial overhead generated by Algorithm 1 is accounted to the run time of the flux evaluation. All performance benchmarks were conducted on an Intel Devil's Canyon platform (i7-4970K, four physical, eight logical cores) with 32 GB of main memory, using a C++ implementation of the presented algorithm.

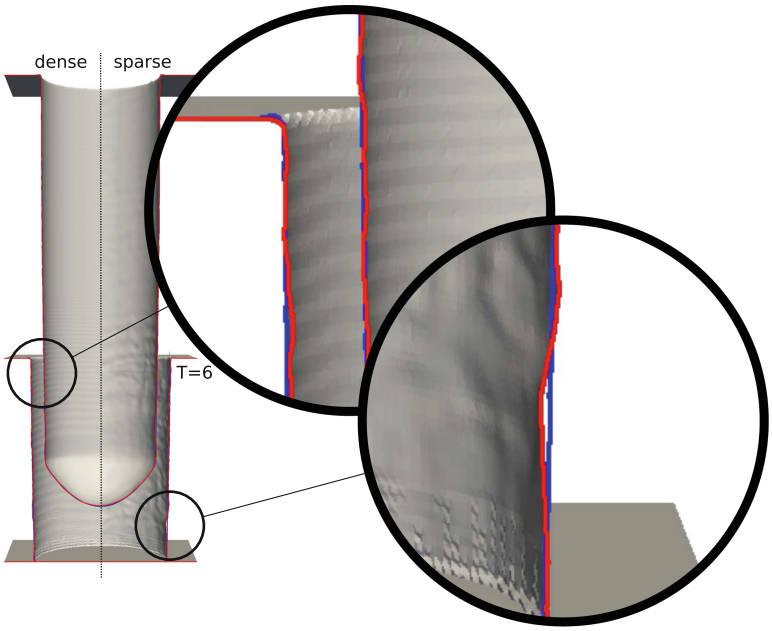


Fig. 4. Comparison of surface positions at times T = [3, 6] for resolution 64. The surface mesh for the dense and sparse flux evaluation is displayed on the left and right half-space, respectively. Two regions are magnified on the right side where the blue and red line correspond to slices of the dense and the sparse evaluation, respectively. (Color figure online)

Figure 5 summarizes the performance differences for resolution 16; the upper plot shows the run time per time step for the dense flux evaluation. The run time at the beginning of the simulation is $\approx 5.5\,\mathrm{s}$ per time step. As soon as the hole has reached the bottom of the bulk material, the number of triangles starts to decrease and consequently the run time per time step drops linearly from T = 3.6 to T = 8. The ratio between flux evaluation (green) and other tasks (red) is ≈ 20 for the whole simulation, emphasizing the dominance of the computational cost for the flux evaluation, even for small domain resolutions. The lower plot in Fig. 5 is analog to the upper plot, but for the sparse flux evaluation. A second y-axis on the right is used to plot two additional properties: the ratio of dense to sparse points (dashed line) and the speedup of the flux evaluation (solid line) over the dense flux evaluation. Throughout the simulation, the dense/sparse ratio is between 2.5 and 6 while the speedup is ≈ 2.0.

Figures 6 and 7 compare the performance for resolution 32 and 64, respectively. With increasing resolution, the dominance of the flux evaluation in terms of run time is also increased, leading to a negligible share of run time for the other tasks in the case of dense flux evaluation. For sparse flux evaluation a dense/sparse ratio of 3 to 14 and 4 to 35 is achieved for resolution 32 and 64, respectively. However, different to resolution 16, the obtained speedups (5 and

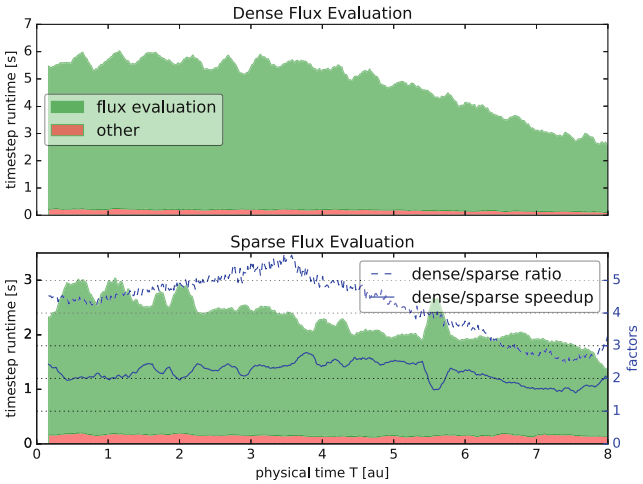


Fig. 5. Performance results for resolution 16. (Color figure online)

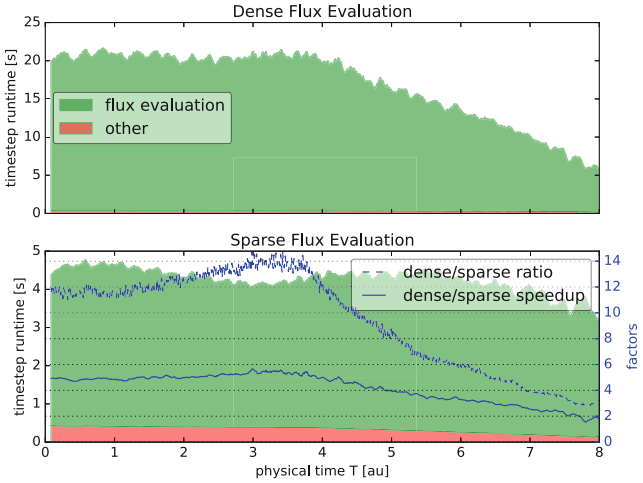


Fig. 6. Performance results for resolution 32. (Color figure online)

8, respectively) are only constant up to T = 3.6, where the hole reaches the bottom of the bulk material. From T = 3.6 to T = 8 the speedups decrease to approximately 2 (following the dense/sparse ratio) keeping the total run time per time step approximately constant up to T = 6.5.

The "gap factor" between achieved and potential speedup (i.e., dense/sparse ratio) is higher for large meshes and ranges from ≈ 2 for resolution 16, to ≈ 4 for resolution 64 before the hole reaches the bottom. When approaching T = 8, all three tested resolutions converge to a speedup of ≈ 2.

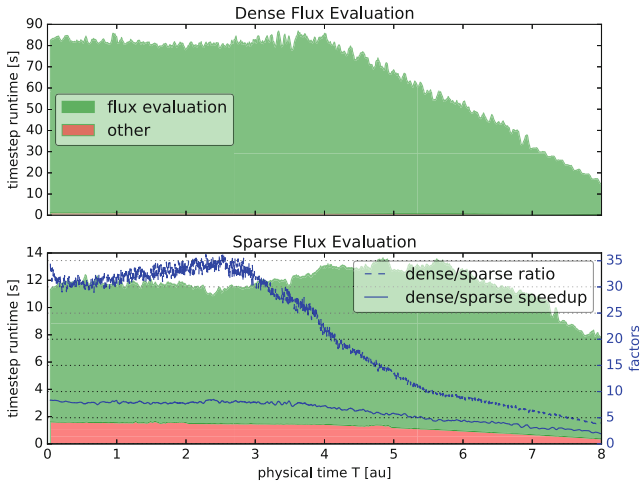


Fig. 7. Performance results for resolution 64. (Color figure online)

5 Summary

We presented a method to reduce the number of necessary evaluations of a surface speed model in each time step of the simulation of a dynamic surface. A sparse point-set and corresponding neighborhoods are constructed using an iterative partitioning scheme. The surface speed model is only evaluated for this sparse point-set. The variable limits for the allowed distance between sparse points enables to balance between computational complexity and accuracy in a robust way. Furthermore, a linear interpolation between the sparse points is approximated by diffusing the result of a constant extrapolation in the neighborhoods using the error smoothing properties of the Jacobi method.

Using a cylindrical hole with a directed vertical source as a generic etching simulation test case, inspired by etching processes arising in semiconductor fabrication, we compare the results against a dense evaluation of the surface speed. Deviations in the surface position are below 3 level-set cells for all tested configurations. The achieved speedups range from 2 for the lowest resolution up to 8 for the highest resolved surface. The speedups are tracked during all time steps of the simulations starting from thick initial geometries to very thin geometries at the end of the simulated physical process.

The method can be adapted to specific application requirements via a freely definable refinement condition for the iterative partitioning scheme. We used a refinement condition based on fixed thresholds of local deviations in surface normal direction and surface speed.

Acknowledgment. The financial support by the *Austrian Federal Ministry of Science, Research and Economy* and the *National Foundation for Research, Technology and Development* is gratefully acknowledged.

A Algorithm Subroutines

Algorithm 2. Recursive flagging and refinement of patches.

```
Function FlagNeighborhood(i, i_parent, i_prev, d_path, d_max):
    d_max_local = distTarget[i];
    if withdrawn[i] AND d_max >= d_path AND d_max_local > d_path AND
    distance[i] > d_path then
        touched[i] = true;
        parent[i] = i_parent;
        distance[i] = d_path;
        foreach i_ne in edgeNeighbors[i] do
            if i_ne != i_prev then
                FlagNeighborhood(i_ne, i_parent, i, d_path + 1, d_max)
    else
        SetNeighbors(i, i_parent);
Function FlagTriangles(indices, d_max):
    touched[] = false;
    d_path = 0;
    numNewPatches = 0;
    foreach i in indices do
        if !touched[i] and withdrawn[i] then
            ++numNewPatches;
            active[i] = touched[i] = reflagged[i] = true;
            parent[i] = i;
            distance[i] = d_path;
            foreach i_ne in edgeNeighbors[i] do
                FlagNeighborhood(i_ne, i, i, d_path + 1, d_max)
    return numNewPatches
Function RefinePatch(i_active, d_max):
    count = Withdraw(i_active, d_max/2)
    if count == 0 then
        return 0
    else
        UnSetAllNeighbors(i_active)
        numNewPatches = FlagTriangles(patches[i_active].patchIndices,
        d_max)
        RebuildNeighbors(i_active)
        UnWithdraw(i_active)
    return numNewPatches
```

Algorithm 3. Helper functions for sparse neighbor handling.

```
Function SetNeighbors(i, i_active):
    if parent[i] != −1 and parent[i] != i_active then
        sparseNeighbors[parent[i]].insert(i_active);
        sparseNeighbors[i_active].insert(parent[i]);
Function UnSetAllNeighbors(i_active):
    foreach i_ns in sparseNeighbors[i_active].activeNeighbors do
        sparseNeighbors[i_ns].erase(i_active);
        sparseNeighbors[i_active].erase(i_ns);
Function RebuildNeighbors(i_active):
    foreach i in patches[i_active].patchIndices do
        if !withdrawn[i] then
            foreach i_ne in edgeNeighbors[i] do
                SetNeighbors(i_ne, i_active);
```

Algorithm 4. Helper functions for withdrawal and building patch information.

```
Function Withdraw(i_active, d):
    count = 0;
    foreach i in patches[i_active].patchIndices do
        if distance[i] > d then
            withdrawn[i] = true;
            distance[i] = d_max0;
            parent[i] = -1;
            ++count;
    return count
Function UnWithdraw(i_active):
    foreach i in patches[i_active].patchIndices do
        withdrawn[i] = false;
Function RebuildPatches():
    patches.clear();
    for i = 0 ... N_tri − 1 do
        if parent[i] == -1 then
            patches[parent[i]].insert(i);
```

References

1. Bronshtein, I., Semendyayev, K., Musiol, G., Mühlig, H.: Handbook of Mathematics, vol. 5. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72122-2
2. Cheong, H.W., Lee, W.H., Kim, J.W., Kim, W.S., Whang, K.W.: A study on reactive ion etching lag of a high aspect ratio contact hole in a magnetized inductively coupled plasma. Plasma Sources Sci. Technol. **23**(6), 065051 (2014)
3. Embree. https://embree.github.io/
4. Hoetzlein, R.K.: GVDB: raytracing sparse voxel database structures on the GPU. In: Proceedings of High Performance Graphics, pp. 109–117 (2016)
5. Lee, C., Dolbow, J., Mucha, P.J.: A narrow-band gradient-augmented level set method for multiphase incompressible flow. J. Comput. Phys. **273**, 12–37 (2014)
6. Manstetten, P., Weinbub, J., Hössinger, A., Selberherr, S.: Using temporary explicit meshes for direct flux calculation on implicit surfaces. Procedia Comput. Sci. **108**, 245–254 (2017)
7. Museth, K.: VDB: high-resolution sparse volumes with dynamic topology. ACM Trans. Graph. **32**(3), 27:1–27:22 (2013)
8. OpenVDB. http://www.openvdb.org/
9. Silvaco Inc: Victory Process - 3D Process Simulator. http://www.silvaco.com/products/tcad/process_simulation/victory_process
10. Wald, I., Woop, S., Benthin, C., Johnson, G.S., Ernst, M.: Embree: a kernel framework for efficient CPU ray tracing. ACM Trans. Graph. **33**(4), 143 (2014)