DOI: 10.1111/cgf.13449

COMPUTER GRAPHICS *forum*
Volume 00 (2018), number 0 pp. 1–13

An Adaptive Multi-Grid Solver for Applications in Computer Graphics

Misha Kazhdan[1] and Hugues Hoppe[2]

[1]Computer Science Department, Johns Hopkins University, Baltimore, MD, USA
misha@cs.jhu.edu
[2]Google Inc., Seattle, WA, USA
hhoppe@gmail.com

Abstract

A key processing step in numerous computer graphics applications is the solution of a linear system discretized over a spatial domain. Often, the linear system can be represented using an adaptive domain tessellation, either because the solution will only be sampled sparsely, or because the solution is known to be 'interesting' (e.g. high frequency) only in localized regions. In this work, we propose an adaptive, finite elements, multi-grid solver capable of efficiently solving such linear systems. Our solver is designed to be general-purpose, supporting finite elements of different degrees, across different dimensions and supporting both integrated and pointwise constraints. We demonstrate the efficacy of our solver in applications including surface reconstruction, image stitching and Euclidean Distance Transform calculation.

Keywords: numerical analysis, matting & compositing, surface reconstruction

ACM CCS: I.3 Computing methodologies → Computer graphics

1. Introduction

Solving linear systems is a fundamental step in numerous applications in image- and geometry-processing. For many such applications, the solution does not need to be fully resolved everywhere—either because the solution is only evaluated sparsely, or because the solution is low-frequency outside of a sparse region. For example, in surface reconstruction applications, the implicit function is only evaluated near the input samples. And in image-stitching applications, the correction is evaluated everywhere but is only high frequency near the seams.

Previous works have addressed this problem using a quadtree/octree to adapt the discretization, and using finite-differences to define a linear system over the adaptively discretized domain. In this work, we extend these approaches in two ways: (1) We describe a finite-elements formulation supporting general partial differential equations, and (2) we leverage the hierarchical structure of the quadtree/octree to design an efficient multi-grid solver.

Starting with the hierarchical solver presented by [KH13] for solving the screened-Poisson equation in 3D, we generalize the solver in several ways, including support for general symmetric positive-definite (SPD) systems:

- in spaces of arbitrary dimension,
- discretized using finite elements of arbitrary degree,
- involving arbitrary function derivatives,
- with both pointwise and integrated constraints.

We also generalize the solver to use a standard V-cycle solver, with temporally blocked Gauss–Seidel relaxation for better runtime performance.

Together, these generalizations provide an efficient and general-purpose solver capable of solving a large class of linear systems in different dimensions. We demonstrate the effectiveness of the solver in applications including surface reconstruction (using both the Laplacian and bi-Laplacian), image-stitching (using mixed-degree finite elements) and the computation of an approximate Euclidean Distance Transform (EDT). In each application, we have found that our general-purpose solver outperforms state-of-the-art methods, providing a solution in less time and less memory.

We begin our discussion by reviewing related work on adapted solvers in general (Section 2), and then focusing on the Poisson Reconstruction solver (Section 3). We then describe our generalization of the Poisson Reconstruction solver (Section 4) and present a method for handling the ringing that occurs due to the abrupt

changes in resolution (Section 5). We demonstrate the utility of our solver in several graphics applications (Section 6) and conclude by summarizing our work and pointing to potential directions for future research (Section 7).

2. Related Work

Much of the related work on adaptive solvers has focused on the problem of solving the Poisson equation in three-dimensional fluid simulations in order to enforce the incompressibility conditions (e.g. [Pop03, LGF04, FOK05, CGFO06]), though applications in other areas have included the solution of the Poisson equation for large image stitching [Aga07] and the solution of Laplacian and bi-Laplacian systems for surface reconstruction [KBH06, MGD*10, CT11, KH13]. In these contexts, solving over a regular grid is impractical as either the grid needs to be made too coarse to capture the fine details or the linear system becomes too large to be solved in practical time/memory.

Perhaps the most direct approach is to use tetrahedral meshes to tessellate space [FOK05, KFCO06, CGFO06, BXH10, MGD*10]. Since the size of the tets can adapt to the regions of interest, the size of the system can be significantly reduced (e.g. adapting to a two-manifold in three-space reduces the complexity from $O(N)$ to $O(N^{2/3})$, with N the number of cells in a regular tessellation). A further advantage of using tetrahedral meshes is that the faces of the tets can be aligned to a triangle mesh, making it easier to enforce boundary conditions. (Later work by Batty *et al.* [BXH10] shows that this is not a requirement, enabling less remeshing for simulations with dynamic boundaries.)

In the context of shallow water simulations, tall cells have been used to discretize the linear system over a regular 2D grid [KM90, OH95]. In addition to reducing the dimensionality of the linear system, this approach has the advantage of using a regular grid as a data structure, reducing neighbour lookup to simple indexing (and thereby enabling better memory utilization). This approach has been extended to shallow water simulations over general surfaces [WMT07] and has been further adapted using hybrid 2D/3D techniques that partition the tall cells in regions of interest [IGLF06, TRS06, CM11]. However, a limitation of this approach is that it does not generalize to systems in which adaptivity is desired along multiple axes.

A more general strategy is to partition space using an adaptive octree [Pop03, LGF04, KBH06, CT11, KH13, Bat17]. Similar to tetrahedral meshing, these approaches allow the discretization to adapt to regions of interest, thereby reducing the dimensionality of the system. In addition, the semi-regular structure of the tree facilitates neighbour lookup through efficient data structures [Sam90] and Z-curve hashing [WS93].

Recent work has also leveraged an octree hierarchy to attain an efficient multi-grid solver for the Poisson equation [SABS14, FWD14], defining a nesting set of suboctrees by successively trimming off nodes at finer resolutions of the tree. Setaluri *et al.* follow the finite-volumes discretization of [LGF04] to define the Laplace operator while Ferstl *et al.* use first-order B-splines to define a finite-elements discretization. Both approaches are restricted to the solution of the Laplace operator—the finite-volumes discretization does not extend to higher order derivatives and the finite-elements discretization relies on the interpolatory property of first-order B-splines to remove 'hanging' vertices.

In contrast, the earlier work of Kazhdan *et al.* [KBH06] (and the follow-on work by Kazhdan and Hoppe [KH13]) uses a more general finite-elements formulation to define a multi-grid solver, allowing for both pointwise and integrated constraints and supporting derivatives of arbitrary degree. We review this approach in the next section.

3. The Poisson Reconstruction Solver

The solver described by Kazhdan *et al.* [KBH06] (and extended in [KH13]) is designed to reconstruct a watertight surface from an oriented point set. The authors reduce the problem of surface reconstruction to the solution of a Poisson equation whose right-hand side encodes the divergence of the surface normal field, showing that the solution gives the coefficients of the indicator function of the shape's interior. To do this, the authors (1) adapt an octree to the points, (2) define a function space over the octree and (3) solve a hierarchical system to obtain the indicator function.

3.1. Finite elements

Given an octree $\mathcal{O}$, the authors define a function space by associating a second-order B-spline with every octree node. Specifically, given a node $o \in \mathcal{O}$, with left/bottom/back corner c_o and width w_o, the authors set $\Phi_o^2 : \mathbb{R}^3 \to \mathbb{R}$ to be the trivariate second-order B-spline centred and scaled with o:

$$\Phi_o^2(q) = \Phi^2\left(\frac{q^x - c_o^x}{w_o}\right) \cdot \Phi^2\left(\frac{q^y - c_o^y}{w_o}\right) \cdot \Phi^2\left(\frac{q^z - c_o^z}{w_o}\right)$$

with $\Phi^2 : \mathbb{R} \to \mathbb{R}$ the univariate second-order B-spline:

$$\Phi^2(s) = \begin{cases} \frac{1}{2} + s + \frac{1}{2}s^2 & \text{if } s \in [-1, 0] \\ \frac{1}{2} + s - s^2 & \text{if } s \in [0, 1] \\ 2 - 2s + \frac{1}{2}s^2 & \text{if } s \in [1, 2] \\ 0 & \text{otherwise.} \end{cases}$$

Using these functions, the Laplacian matrix, $\mathbf{L}$, is discretized by setting the coefficient corresponding to the pair of nodes $o, \tilde{o} \in \mathcal{O}$ to be the integral:

$$\mathbf{L}_{o,\tilde{o}} = \int_{[0,1]^3} \Delta\Phi_o^2(q) \cdot \Phi_{\tilde{o}}^2(q)\, \mathrm{d}q.$$

Note that because the functions $\{\Phi_o\}$ are compactly supported, the matrix $\mathbf{L}$ will be sparse, with $\mathbf{L}_{o,\tilde{o}} = 0$ whenever the supports of Φ_o and $\Phi_{\tilde{o}}$ do not overlap.

Kazhdan and Hoppe [KH13] extend the solver to support pointwise interpolation. This is done by augmenting the Laplacian with a screening term:

$$\mathbf{S}_{o,\tilde{o}} = \sum_{p \in \mathcal{P}} \omega(p) \cdot \Phi_o^2(p) \cdot \Phi_{\tilde{o}}^2(p),$$

where $\mathcal{P}$ is the point set and $\omega : \mathcal{P} \to \mathbb{R}^{\geq 0}$ is a weighting function assigning an importance to each point constraint.

3.2. Multi-grid

To solve the linear system $\mathbf{M}X = B$, with X the solution coefficients, B the right-hand side encoding the divergence of the normal field and $\mathbf{M} = \mathbf{L}$ (or $\mathbf{M} = \mathbf{L} + \mathbf{S}$ in the screened system), Kazhdan *et al.* [KBH06] use the octree hierarchy to define a (cascadic) multi-grid solver. Specifically, given the target number of relaxation iterations, ν, the solution X is obtained using the **LogLinearCascadicSolver[2006]** algorithm.

LogLinearCascadicSolver[2006]($\mathbf{M}$, B, ν)

```
1  X ← 0                                                          initialize
2  for ℓ ∈ {0, ..., Λ − 1}
3    for ℓ̃ ∈ {0, ..., ℓ − 1}: B^ℓ ← B^ℓ − M^ℓ_ℓ̃(X^ℓ̃)              adjust
4    if( ℓ ≠ 0)  X^ℓ ← Relax( M^ℓ_ℓ, X^ℓ, B^ℓ, ν )                relax
5    else        X^ℓ ← Solve( M^ℓ_ℓ, B^ℓ )                        solve
6  return X
```

The algorithm performs a coarse-to-fine iteration through the levels of the tree (line 2), adjusts the constraints at the given level to discount those constraints met at coarser resolutions (line 3) and refines the solution (lines 4 and 5).

Here Λ is the depth of the octree, X^ℓ (respectively, B^ℓ) is the subvector of X (respectively, B) indexed by nodes at level ℓ, $\mathbf{M}^\ell_{\tilde{\ell}}$ is the submatrix of the (screened) Laplacian with columns indexed by nodes at level ℓ and rows indexed by the nodes at level $\tilde{\ell}$, **Relax** is the operator performing ν iterations of a conjugate-gradients solver and **Solve** is an operator computing the exact solution of a symmetric, positive, semi-definite, linear system.

The hierarchical solver is made more efficient in the work of Kazhdan and Hoppe [KH13] who observe that while one cannot prolong the entirety of the coarse solution without fully refining the octree, it suffices to prolong only the part of the coarser solution that is 'overlapped' by the finer level. Specifically, defining two nodes o and $\tilde{o}$ to be *overlapped* when the functions Φ_o and $\Phi_{\tilde{o}}$ have overlapping support, the authors propose refining the octree so that when a node is in the tree at level ℓ, all overlapping nodes at level $\ell - 1$ are also in the tree. The modified cascadic solver is summarized in the **LinearCascadicSolver[2013]** algorithm.

LinearCascadicSolver[2013]($\mathbf{M}$, B, ν)

```
1  X ← 0                                                          initialize
2  for ℓ ∈ {0, ..., Λ − 1}
3    B^ℓ ← B^ℓ − M^ℓ_{ℓ−1}(P^{ℓ−1})                               adjust
4    if( ℓ ≠ 0 ) X^ℓ ← Relax( M^ℓ_ℓ, X^ℓ, B^ℓ, ν )                relax
5    else        X^ℓ ← Solve( M^ℓ_ℓ, B^ℓ )                        solve
6    P^ℓ ← X^ℓ + P^ℓ_{ℓ−1}(P^{ℓ−1})                               prolong
7  return X
```

After solving at a given level (lines 4 and 5), the algorithm upsamples the accumulated solution from the coarser levels and combines it with the solution at the current level (line 6). This way, when adjusting the constraints at a given level only the prolongation from the previous level needs to be considered (line 3). For an octree with $|\mathcal{O}|$ nodes, this reduces the complexity of the cascadic solve from $O(|\mathcal{O}| \cdot \log|\mathcal{O}|)$ to $O(|\mathcal{O}|)$.

Here the prolongation operator $\mathbf{P}^\ell_{\ell-1}$ upsamples a solution at level $\ell - 1$ into a solution at level ℓ using the tensor-product of the standard second-order B-spline prolongation stencil, $(\frac{1}{4}, \frac{3}{4}, \frac{3}{4}, \frac{1}{4})$. (At level $\ell = 0$, the values $\mathbf{M}^\ell_{\ell-1}(P^{\ell-1})$ and $\mathbf{P}^\ell_{\ell-1}(P^{\ell-1})$ are assumed to be zero.)

4. Generalizing the Solver

In this work, we generalize the hierarchical solver from [KBH06] and [KH13] to support the solution of a much larger class of linear systems. Specifically, we develop a solver that (1) uses finite elements of arbitrary degrees, (2) solves general SPD systems, (3) supports general integral and pointwise constraints and (4) works in arbitrary dimensions. In addition, we modify the solver to support full V-cycle iterations and to use Gauss–Seidel relaxation.

4.1. Extending the system

4.1.1. *Finite -element degree*

The solver in [KBH06] associates a trivariate second-order B-spline to each node of the tree. We extend the formulation by supporting the use of B-splines of any degree. This allows for trading off between system sparsity (lower degree) and smoothness (higher degree). In doing so, we also replace the prolongation stencils to conform with the chosen degree.

Even-degree/dual elements. Extending to B-splines of even-degree is straightforward as these B-splines are centred at the centres of grid cells.

Odd-degree/primal elements Extending to B-splines of odd-degree is more nuanced as these B-splines are centred at the corners of grid cells. To avoid redundant indexing, we associate each octree node with a single B-spline, centred at the left, bottom, back corner. We also extend the octree to partition the cube $[0, 2] \times [0, 2] \times [0, 2]$. This ensures that our discretization has the ability to represent any B-spline centred at a corner of a grid cell. Note that though the new octree partitions a larger domain, integration is restricted to the domain $[0, 1] \times [0, 1] \times [0, 1]$.

In this case, we make a distinction between the logical depth of the tree (which is exposed to the user) and the actual depth of the tree (which is equal to the logical depth plus one, and is used for internal book-keeping). For the remainder of the paper, 'depth' will be understood to refer to the logical depth.

Mixed-degree elements We also support B-splines of mixed-degree, centreing these on the cells/faces/edges/vertices of the grid. As with odd-degree B-splines, we use the left/bottom/back association and extend the octree to partition a larger cube. For example, using B-splines of mixed-degree $D = \{2, 1, 2\}$, we associate the function Φ^D_o to the centre of the bottom face of octree cell o, with:

$$\Phi^D_o(q) = \Phi^2\left(\frac{q^x - c^x_o}{w_o}\right) \cdot \Phi^1\left(\frac{q^y - c^y_o}{w_o}\right) \cdot \Phi^2\left(\frac{q^z - c^z_o}{w_o}\right),$$

where $\Phi^1 : \mathbb{R} \to \mathbb{R}$ is the univariate first-order B-spline:

$$\Phi^1(s) = \begin{cases} 1+s & \text{if } s \in [-1,0] \\ 1-s & \text{if } s \in [0,1] \\ 0 & \text{otherwise.} \end{cases}$$

The advantage of supporting mixed-degree B-splines is that partial derivatives of uniform-degree B-splines can be expressed as the linear combinations of mixed-degree B-splines. The coefficients of these partial derivatives are given in terms of finite-differences of the original coefficients and are naturally stored on a staggered grid [HW65].

4.1.2. *Prescribing a linear system*

The solver in [KBH06] (and [KH13]) defines a (screened) Poisson equation, with coefficients expressed in terms of the integrated dot-product of gradients (and weighted sum of the product of pointwise evaluations). We extend this formulation to support the solution of a more general class of linear systems. Specifically, given $D = \{d_x, d_y, d_z\}$ – the degrees of the system B-splines along the x, y and z axes – we denote by $\mathcal{D} = [0, d_x] \times [0, d_y] \times [0, d_z]$ the set of all possible partial derivatives and support the construction of both integrated and sampled coefficients that incorporate all bilinear combinations of the $|\mathcal{D}|$ derivatives.

Integrated system We support the construction of general integrated linear systems by allowing the user to prescribe any symmetric, positive-semi-definite matrix, $\mathbf{D} \in \mathbb{R}^{|\mathcal{D}|\times|\mathcal{D}|}$. We then define the associated linear system, $\mathbf{L} \in \mathbb{R}^{|\mathcal{O}|\times|\mathcal{O}|}$, by setting:

$$\mathbf{L}_{o,\tilde{o}} = \sum_{(\boldsymbol{i},\tilde{\boldsymbol{i}})\in\mathcal{D}\times\mathcal{D}} \mathbf{D}_{\boldsymbol{i},\tilde{\boldsymbol{i}}} \cdot \int_{[0,1]^3} \partial^{\boldsymbol{i}} \Phi_o^D(q) \cdot \partial^{\tilde{\boldsymbol{i}}} \Phi_{\tilde{o}}^D(q)\, \mathrm{d}q,$$

where for $\boldsymbol{i} = \{i, j, k\}$, we have

$$\partial^{\boldsymbol{i}} \Phi(p) \equiv \frac{\partial^i}{\partial x^i} \frac{\partial^j}{\partial y^j} \frac{\partial^k}{\partial z^k} \Phi(p).$$

Pointwise system Given a point set $\mathcal{P}$, we support the construction of general pointwise linear systems by allowing the user to prescribe a function $\boldsymbol{D} : P \to \mathbb{R}^{|\mathcal{D}|\times|\mathcal{D}|}$ returning a positive-semi-definite matrix. We then define the associated linear system, $\mathbf{S} \in \mathbb{R}^{|\mathcal{O}|\times|\mathcal{O}|}$, by setting:

$$\mathbf{S}_{o,\tilde{o}} = \sum_{(\boldsymbol{i},\tilde{\boldsymbol{i}})\in\mathcal{D}\times\mathcal{D}} \sum_{p\in\mathcal{P}} \boldsymbol{D}_{\boldsymbol{i},\tilde{\boldsymbol{i}}}(p) \cdot \partial^{\boldsymbol{i}} \Phi_o^D(p) \cdot \partial^{\tilde{\boldsymbol{i}}} \Phi_{\tilde{o}}^D(p).$$

4.1.3. *Imposing constraints*

In [KBH06], the constraint is given in terms of the integrated products of the gradients of the B-splines with a vector field describing the normals of the point-cloud. This is extended in [KH13] to also constrain the solution to evaluate to 0.5 at the point samples. We extend this formulation to support the prescription of a more general family of integrated and pointwise constraints.

Integrated constraints As above, we let $\mathcal{D} = [0, d_x] \times [0, d_y] \times [0, d_z]$ denote the set of all possible partial derivatives of the system B-splines. We support the construction of general integrated constraints by allowing the user to prescribe the coefficients, $F \in \mathbb{R}^{|\mathcal{O}|}$, of a function with respect to B-splines of degree $\tilde{D} = \{\tilde{d}_x, \tilde{d}_y, \tilde{d}_z\}$, as well a matrix, $\mathbf{M} \in \mathbb{R}^{|\mathcal{D}|\times|\tilde{\mathcal{D}}|}$. Then, given an octree node $o \in \mathcal{O}$, we increment the oth coefficient of the constraint vector by setting:

$$B_o \mathrel{+}= \sum_{\tilde{o}\in\mathcal{O}} F_{\tilde{o}} \cdot \sum_{(\boldsymbol{i},\tilde{\boldsymbol{i}})\in\mathcal{D}\times\tilde{\mathcal{D}}} \mathbf{M}_{\boldsymbol{i},\tilde{\boldsymbol{i}}} \cdot \int_{[0,1]^3} \partial^{\boldsymbol{i}} \Phi_o^D(q) \cdot \partial^{\tilde{\boldsymbol{i}}} \Phi_{\tilde{o}}^{\tilde{D}}(q)\, \mathrm{d}q.$$

Pointwise constraints We support the construction of general pointwise constraints by allowing the user to prescribe a function $\boldsymbol{F} : P \to \mathbb{R}^{|\mathcal{D}|}$. Then, given an octree node $o \in \mathcal{O}$, we increment the oth coefficient of the constraint vector by setting:

$$B_o \mathrel{+}= \sum_{\tilde{o}\in\mathcal{O}} \sum_{\boldsymbol{i}\in\mathcal{D}} \sum_{p\in\mathcal{P}} \boldsymbol{F}_{\boldsymbol{i}}(p) \cdot \partial^{\boldsymbol{i}} \Phi_{\tilde{o}}^D(p).$$

4.1.4. *Dimensionality*

The solver in [KBH06] was designed for linear systems in 3D. We extend the formulation by supporting the design of linear systems in any dimensions.

Conceptually, this is straightforward to do because multi-variate B-splines are defined as tensor-products of univariate B-splines. Thus, function integration and evaluation can be performed by separately integrating/evaluating across the different dimensions and then multiplying. (Similarly, multi-variate prolongation stencils are described as the tensor-products of univariate stencils.)

We make this efficient in practice using several techniques.

***m*-Dimensional windows** We define an *m-dimensional window* to be a wrapper for m-dimensional arrays of size $w_1 \times \cdots \times w_m$ (with $w_i \in \mathbb{Z}^{\geq 0}$ a compile-time constant). The window implements functionality returning the $(m-1)$-dimensional slices of size $w_1 \times \ldots \times w_{m-1}$ and supports recursive iteration over the window dimensions to apply a computation kernel at the individual elements.

Neighbour lookup As most of the computation in defining and solving the linear system requires finding the neighbours of a node, we design a *neighbour-key* data structure to amortize the cost of neighbour lookup. Given an octree $\mathcal{O}$ of depth Λ and given a target neighbour window $\mathcal{W} = [-l_1, r_1] \times \cdots \times [-l_m, r_m]$ (with, $l_i, r_i \in \mathbb{Z}^{\geq 0}$ compile-time constants), we define a neighbour-key to be an array of Λ windows of size $\mathcal{W}$. Since the $\mathcal{W}$ neighbours of a node are contained in the children of the $\mathcal{W}$ neighbours of the node's parent, we recursively set the neighbour-key by setting the parent's neighbour and using those to set the neighbours of the child.

The advantage of this approach is that if we traverse the octree in a coherent (e.g. Morton) order, the node's parent's neighbours will likely have been set in a previous neighbour lookup and we will not need to traverse to the root of the tree to compute neighbours. Thus, the amortized cost of computing the neighbours within a fixed window is reduced to $O(1)$.

Template specialization As the size of the windows we use is determined by the degrees of the B-splines, which are known at compile-time, we implement the windowed computations using C++ templates specifying the degrees of the B-splines along the different axes. The templating of dimension of the system makes it easier for the compiler to inline the nested iterations (rather than using recursive function calls to iterate over the windows) while the templating of the window dimensions enables loop-unrolling, both of which improve the runtime performance. (Technically, we use variadic templates with integer parameters: The number of parameters defines the dimension of the Euclidean space and the values of the parameters give the degrees along the individual axes.)

We note that with the exception of the specialization of the template code for the case of dimension-one (to terminate the recursive iteration over the window slices) the only code that we specialize is the isosurface extraction, which is only implemented for 3D.

4.2. Modifying the solver

We modify the solver in two ways. We improve the accuracy of the solver by extending it to support a full V-cycle pass and we improve the memory efficiency of the solver by replacing the conjugate-gradients solver with a Gauss–Seidel solver.

V-Cycle Solver

We extend the solver to support V-cycle iterations by adjusting the constraints at level ℓ using the computed solutions at both finer and coarser resolutions. As with the extension in [KH13], this can be made efficient when the octree is refined so that when a node is in the tree, all coarser overlapped nodes are also in the tree.

Algorithm **VCycleSolver** provides pseudo-code for a solver performing n V-cycles. After relaxing the solution in the restriction phase (line 5), the algorithm downsamples the accumulated (dual) solution from the finer levels and combines it with the (dual) solution at the current level (line 6). Then, when computing the residual constraint (lines 4, 7 and 11), both the finer and coarser solutions are used to adjust the constraints. Here, the restriction operator $\mathbf{R}_\ell^{\ell-1}$ is the transpose of $\mathbf{P}_{\ell-1}^{\ell}$.

VCycleSolver($\mathbf{M}$, B, n, ν)

1	$X, P, R \leftarrow 0$	*initialize*
2	`for` $\sigma \in \{1, \ldots, n\}$	
	//Fine-to-coarse relaxation	
3	`for` $\ell \in \{\Lambda - 1, \ldots, 1\}$	
4	$\widehat{B}^\ell \leftarrow B^\ell - R^\ell - \mathbf{M}_{\ell-1}^{\ell}(P^{\ell-1})$	*adjust*
5	$X^\ell \leftarrow$ **Relax**($\mathbf{M}_\ell^\ell$, X^ℓ , $\widehat{B}^\ell$, ν)	*relax*
6	$R^{\ell-1} \leftarrow \mathbf{M}_\ell^{\ell-1}(X^\ell) + \mathbf{R}_\ell^{\ell-1}(R^\ell)$	*restrict*
	//Apply direct solve at coarsest resolution	
7	$\widehat{B}^0 \leftarrow B^0 - R^0$	*adjust*
8	$X^0 \leftarrow$ **Solve**($\mathbf{M}_0^0$, $\widehat{B}^0$)	*solve*
9	$P^0 \leftarrow X^0$	*prolong*
	//Coarse-to-fine relaxation	
10	`for` $\ell \in \{1, \ldots, \Lambda - 1\}$	
11	$\widehat{B}^\ell \leftarrow B^\ell - R^\ell - \mathbf{M}_{\ell-1}^{\ell}(P^{\ell-1})$	*adjust*
12	$X^\ell \leftarrow$ **Relax**($\mathbf{M}_\ell^\ell$, X^ℓ, $\widehat{B}^\ell$, ν)	*relax*
13	$P^\ell \leftarrow X^\ell + \mathbf{P}_{\ell-1}^{\ell}(P^{\ell-1})$	*prolong*
14	`return` X	

Gauss–Seidel Smoothing

We also modify the relaxation step (lines 5 and 12 of Algorithm **VCycleSolver**) to use Gauss–Seidel iterations instead of conjugate gradients. This allows us to leverage temporal blocking [Pfe63, DHK*00, CK11] to sequentially stream in slices of the octree, maintaining a small working window in memory as we perform the relaxation. This improves cache coherence and reduces the overall memory usage by allowing us to construct the system matrix slice-by-slice (i.e. without ever having to store the whole matrix in memory). We further parallelize the relaxation by using multi-colour Gauss–Seidel iterations within each slice, with the number of colours determined by the support/degree of the finite elements.

5. Ringing

As the function space we use is adaptive, our approach can only represent functions that are high frequency where the grid is adaptively refined. Our initial motivation was to consider applications which only evaluate the solution in these high-frequency regions. However, a broader set of applications can be considered if we allow the solution to be evaluated everywhere.

In this section, we describe the ringing problem that arises when evaluating away from the adaptively refined regions and provide a heuristic for resolving this problem.

5.1. Ringing in multi-grid

To motivate the problem, we consider a simple 1D case in which we seek the function whose gradients best fit the gradients of a depth-5 Heaviside function. (This is the function whose coefficients are equal to 0 for the first 16 indices and equal to 1 for the second 16 indices, visualized in the bottom right of Figure 1.)

Figure 1 shows a visualization of the solutions obtained at different resolutions. As expected, when using the function space defined at depth 5, we reconstruct the original step function. However, at coarser depths the function spaces can no longer reproduce the higher-frequency components and ringing is visible.

This ringing is not a problem for standard multi-grid solvers because the solution obtained at a coarser resolution is prolonged into the finer resolution space, where subsequent relaxations remove the high-frequency artefacts. However, when using a solver over an adaptive domain, not all of the solution computed at the coarser resolution can be prolonged. In particular, ringing that occurs outside the support of the finer functions is not 'seen' at the finer resolutions and cannot be removed. Figure 2 (left) shows a visualization of this type of ringing, when the tree is only refined near the discontinuity in the step function.

5.2. Adapted successive underrelaxation

We address the ringing problem in two steps. First, we use a V-cycle solver, initially performing fine-to-coarse relaxations (restriction) and then performing coarse-to-fine relaxations (prolongation). Our goal in relaxing at the higher frequencies first is to resolve the high-frequency components of the constraints so that the restricted

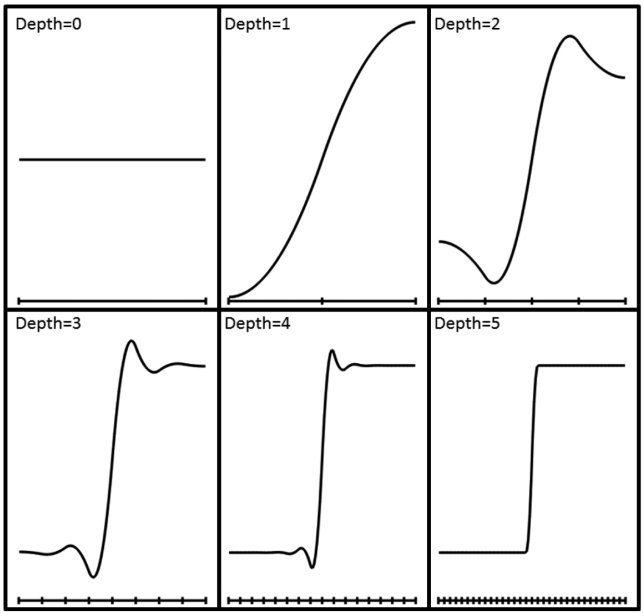

Figure 1: *Using a regular grid to solve for the function whose gradients best-fit the gradients of a depth-5 step function results in ringing artefacts when solved at coarser depths. Using a depth-5 system (bottom right), we correctly reconstruct the step function.*

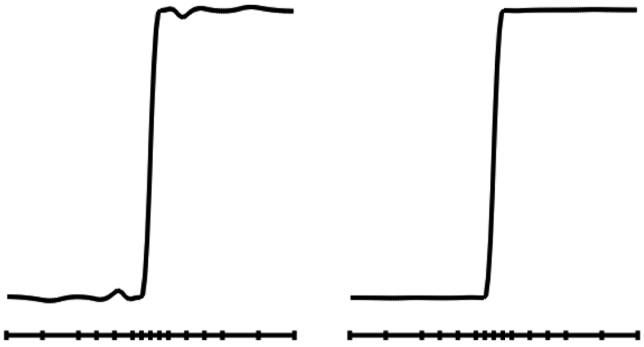

Figure 2: *Visualizations of the solutions obtained using a discretization where the tree is only refined near the discontinuity in the step function: Using a cascadic solver without adapted underrelaxation the solution exhibits significant ringing (left); Using a V-cycle solver with adapted underrelaxations in the restriction phase successfully removes the artefacts (right). Tick marks at the bottom show how the interval is partitioned by the leaf nodes of the adaptive binary tree.*

residual represents a smoother function, and can be solved at coarser resolutions without introducing ringing.

On its own, we have found that using a V-cycle solver produces results that are worse than those obtained using a strictly coarse-to-fine (cascadic) solver because the use of an adaptive grid introduces Dirichlet-like constraints at the higher frequencies. (Using an adaptive solver can be thought of as using a regular solver but fixing the coefficients of nodes outside the adaptive region to zero.) To mitigate these effects, we use adapted successive underrelaxation in the restriction phase to downplay the contribution of nodes near the refinement boundary.

Specifically, given a node $o \in \mathcal{O}$ at depth d, we measure the proximity of o to the boundary by measuring the ratio of integrals:

$$\mu(o) = \frac{\int_{\Omega^d} B_o(p)\,\mathrm{d}p}{\int_{\Omega} B_o(p)\,\mathrm{d}p},$$

with $B_o(p)$ the B-spline associated with node o, Ω the working volume and $\Omega^d \subset \Omega$ the subset contained within nodes at depth d. (As we are using B-splines of degree greater than zero, the support of a B-spline extends beyond the interior of the associated node.) Here the denominator measures the total mass of the B-spline, $B_o(p)$, while the numerator measures the mass contained within the depth-d nodes, so that $\mu(o) = 1$ for nodes interior to the adaptive region and $\mu(o) < 1$ for nodes near the refinement boundary.

We then proceed as with standard Gauss–Seidel, iteratively updating the coefficient X_o by computing the correction term, C_o, for each node n. However, instead of adding the full correction term, we add a fraction:

$$X_o \leftarrow X_o + \alpha \cdot \mu(o)^{\beta} \cdot C_o,$$

where the parameters α and β control the extent of the underrelaxation as a function of the proximity to the boundary. Figure 2 (right) shows the results obtained when solving for the function whose gradients best fit the gradients of the depth-5 Heaviside function using adapted successive underrelaxation during the restriction phase. As the figure shows, this heuristic effectively removes the ringing artefacts.

6. Results

To evaluate our multi-grid solver, we consider three different applications: surface reconstruction, gradient-domain image stitching and EDT calculation. The results described in this section were obtained on a PC running Windows 10, with an Intel Core i7-6700HQ processor and 16 GB of RAM. We used eight Gauss–Seidel relaxations at each hierarchy level and fixed the underrelaxation parameters at $\alpha = 1/8$ and $\beta = 6$ for all evaluations.

6.1. Surface reconstruction

In these applications, the goal is to compute a watertight mesh that fits an input-oriented point set. We consider two implementations, Screened Poisson Surface Reconstruction (SPR) [KH13] and the Smoothed Signed Distance (SSD) Reconstruction [CT11, CT12]. Both proceed by first computing an implicit function discretized over an octree adapted to the input samples, and then extracting an isosurface.

For conciseness, we provide only a small number of reconstruction visualizations in this section. For additional results, please see Figure 9 in the supplemental.

Table 1: *Comparison of running time (seconds) and memory usage (MB) for the original SPR and SSD implementations [Kaz13, CMT11] with the implementations obtained using our adaptive solver at depth 10. Numbers in brackets give the timing for a single-threaded implementation. †Models denoted with a dagger are reconstructed with colour. As in the original implementation of SPR, our implementation of SPR and SSD performs a single cascadic multi-grid pass.*

		Screened Poisson reconstruction						SSD reconstruction			
		Original		Ours (deg. 1)		Ours (deg. 2)		Original		Ours	
	Points	sec.	MB	sec.	MB	sec.	MB	sec.	MB	sec.	MB
Angel	24K	2	118	2	54	3	89	3	114	[4] 3	88
Bunny	354K	18	891	13	379	19	542	34	1287	[34] 13	402
Eagle	778K	49	1239	28	672	43	907	49	1943	[75] 26	667
Eagle†	778K			33	731	48	923	58	2591	[82] 30	695
Tanagra	2M	28	686	17	427	26	577	45	1418	[51] 18	404
King	2M	57	1305	35	877	56	1331	69	2705	[111] 36	913
King†	2M			41	950	62	1361	84	3733	[118] 42	969
Thai Statue	2M	45	1137	28	696	43	1139	72	2310	[85] 29	842
Guinea Pig	2M	36	933	20	479	31	617	75	1583	[64] 23	477
Guinea Pig†	2M			24	519	36	640	80	2217	[69] 28	513
Head	3M	41	885	23	537	35	743	69	1866	[70] 25	568
Head†	3M			27	582	40	767	84	2563	[76] 29	609
Broccoli	3M	71	1581	41	973	64	1134	115	3048	[124] 42	961
Broccoli†	3M			49	1051	71	1376	131	4301	[135] 49	1027
Neptune	4M	37	608	18	387	27	601	163	1343	[50] 19	450
Lucy	5M	45	753	22	502	34	861	161	1725	[68] 25	629
Monk	5M	43	778	22	499	34	770	101	1749	[70] 26	583
Monk†	5M			28	543	39	800	142	2516	[77] 31	627
David	11M	144	2378	77	1453	113	1750	386	5178	[230] 80	1328
Awakening	20M	123	992	43	626	59	880	576	2871	[108] 46	684

Screened Poisson Surface Reconstruction

This work computes the indicator function (i.e. the function that is zero outside of the surface and one inside) using degree-two finite elements by solving a linear system composed of two parts: a Poisson system that seeks a function whose gradients match a smoothed normal field defined by the oriented points, and a screening system that seeks a function evaluating to 0.5 at the point samples.

As our solver extends the solver in [KH13], it too can be used for computing the implicit function. Furthermore, because our solver supports finite-elements systems of any degree, and since the formulation of the Poisson system only requires computing first derivatives, we can implement SPR using degree-one finite elements. (To extract a smooth isosurface, we use the gradients at the octree corners to define piecewise quadratic interpolants along the edges as in [FKG15]. Because degree-one finite elements have derivative discontinuities at the corners, we define the gradients by averaging the left- and right-sided derivatives.)

Table 1 (left) compares the running time and memory usage of our solver with the implementation provided in [Kaz13]. Comparing the performance of the original solver with our degree-two solver, we see that the extension of the adaptive solver to a larger class of linear systems does not come at the cost of slower running times or larger memory usage. (Our implementation runs in 85% of the time and uses 80% of the memory.)

The advantage of a general-purpose solver becomes more pronounced when we compare the original implementation to the implementation using degree-one finite elements. These elements have smaller support and result in a sparser linear system. As Table 1 shows, the sparser system results in significantly improved performance. (Our implementation runs in 55% of the time and uses 60% of the memory.)

Figure 4 (top) shows example close-ups from the reconstructions of the David head, using the original SPR algorithm and our implementation with degree-one and degree-two elements. Despite the lower degree of the finite elements, we did not find a noticeable difference in the quality of the reconstructed surfaces when using lower-degree elements. This is confirmed quantitatively in Figure 3 which gives the root mean squared distance from the input point set to the reconstructed surface, and shows that the original implementation and our implementation using degree-one and degree-two elements produce reconstructions of similar quality.

SSD Reconstruction

This work computes an approximation to the signed EDT (i.e. the function that gives the signed distance from a point in space to the nearest point on the surface) using degree-two finite elements by solving a linear system composed of three parts: a bi-Laplacian system that seeks a function whose gradients are as constant as possible, a value-screening system that seeks a function evaluating

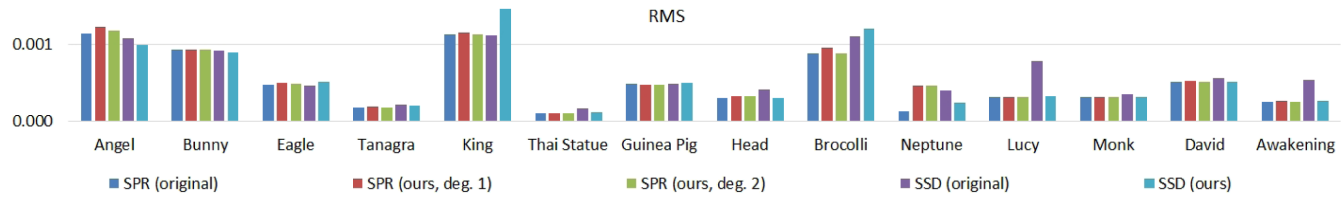

Figure 3: *Comparison of reconstruction accuracy for the original SPR and SSD implementations [Kaz13, CMT11] with the implementations obtained using our adaptive solver at depth 10. Accuracy is measured in RMS and is computed by using half of the points for reconstruction and measuring the one-sided distance from the second half of the points to the reconstructed surface.*

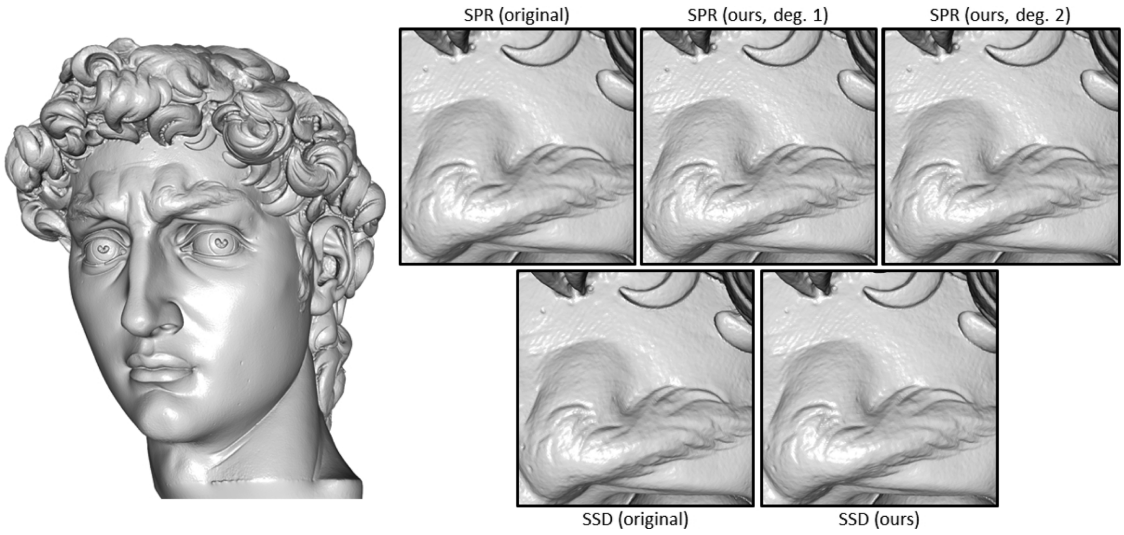

Figure 4: *Reconstructions of David's head (left) at depth 10, comparing the original SPR implementation [Kaz13] (top left), the SPR implementation with our adaptive solver using degree-one elements (top centre), the SPR implementation with our adaptive solver using degree-two elements (top right), the original SSD implementation [CMT11] (bottom left) and the SSD implementation with our adaptive solver (bottom right),*

to 0 at the point samples and a gradient-screening system that seeks a function whose gradients agree with the normals at the point samples.

As our solver supports systems constraining both integrals and values of higher-order derivatives of functions, it can be extended to support the SSD reconstruction algorithm. However, as this system requires the computation of second-order derivatives, our implementation is constrained to use at least degree-two finite elements.

Table 1 (right) compares the running time and memory usage of our solver with the implementation provided in [CMT11]. As the original implementation is single-threaded, we provide running times for our implementation with and without multi-threading. Comparing the performance of the original solver with our implementation, we see that despite the optimized implementation of the authors, our general-purpose solver provides an implementation with significantly improved running times and memory usage. (Our single-threaded implementation runs in 80% of the time and uses 30% of the memory. Our multi-threaded implementation runs in 30% of the time.)

Figure 4 (bottom) shows example close-ups from the reconstructions of the David head, using the original SSD implementation and our adaptive multi-grid implementation using degree-two elements. The figure shows that the two reconstructions are qualitatively similar. This is verified empirically in Figure 3 which shows that the two implementations produce reconstructions with similar quality. (When running the implementation provided in [CMT11] we used a value weight that was five times the default. We found that this produced higher quality results in less time.)

Colour Interpolation

We extend our implementation of SPR and SSD to support colour interpolation by constructing a colour function, $C : [0, 1]^3 \rightarrow [0, 256)^3$, that is evaluated at the vertices of the reconstructed mesh. Our implementation follows the hierarchical push–pull approach for scattered data interpolation [Bur88, GGSC96].

During the construction of the octree, we distribute each input point's colour into the eight corners of the octree node containing the point. The colour is distributed using trilinear interpolation

weights and both the weighted colour and the weights themselves are accumulated at the corners.

Distributing at each level of the octree, we obtain piecewise trilinear functions $c_\ell(p)$ and $w_\ell(p)$ giving the accumulated colour and weight at level ℓ. At deeper levels of the tree (larger ℓ), the functions c_ℓ and w_ℓ provide a finer representation of the colour, but are not well-defined away from the input samples. At coarser levels, the functions provide a smoothed representation of the colour that is defined further away from the input. We take a linear combination of these to get a single colour and weight function, weighting the contributions to give preference to colours/weights at finer resolutions:

$$c(p) = \sum_{\ell=0}^{\Lambda-1} c_\ell(p) \cdot \kappa^\ell \qquad \text{and} \qquad w(p) = \sum_{\ell=0}^{\Lambda-1} w_\ell(p) \cdot \kappa^\ell$$

with Λ the number of levels in the octree and κ the weight (fixed at $\kappa = 32$ in our experiments). Dividing the accumulated colour by the accumulated weights, we get the final colour function:

$$C(p) = c(p)/w(p).$$

Table 1 shows the performance of our degree-one and degree-two SPR implementations and compares the performance of our colour SSD reconstruction with the implementation provided in [CMT11]. Comparing the implementations, we see that all the methods incur a small runtime cost for colour interpolation. However, the original SSD implementation incurs a substantially larger memory cost.

Figure 5 compares close-ups from the colour reconstructions for the Eagle model. As the figure shows, the different methods all produce similar colour reconstructions. This is confirmed quantitatively

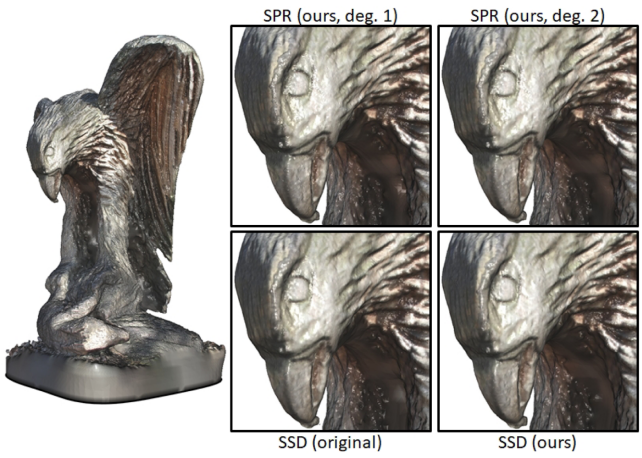

Figure 5: *Colour reconstructions of the Eagle model at depth 10, comparing SPR with our adaptive solver using degree-one elements (top centre), SPR with our adaptive solver using degree-two elements (top right), the original SSD implementation [CMT11] (bottom left) and the SSD reconstruction with our adaptive solver (bottom right).*

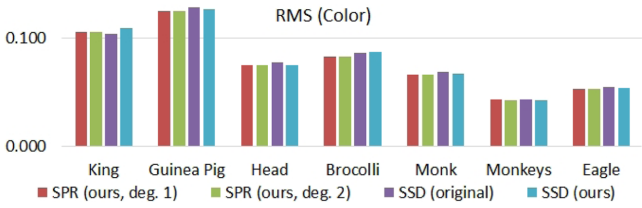

Figure 6: *Comparison of colour reconstruction accuracy for our PSR implementation using degree-one and degree-two elements, as well as the original SSD [CMT11] and our SSD implementations at depth 10. Accuracy is measured in RMS and is computed by using half of the points for reconstruction and measuring the one-sided difference from the colours associated to the second half of the points to the colours at the nearest point on the reconstructed surface.*

in Figure 6 which gives the root-mean squared distance from the colours of the input point set to the colours of the reconstructed surface, and shows that all methods produce colours with comparable quality.

6.2. Gradient-domain image stitching

In this application, the goal is to stitch together a panorama consisting of multiple images by removing discontinuities along seam boundaries. Following the approach of Agarwala [Aga07], stitching can be formulated as the problem of finding the offset function that removes the seams. This function is defined in the gradient-domain by setting the gradients to be equal to zero away from the seams (so that adding the offset leaves the interior unchanged) and setting the gradients to the negative of the composite's gradient across the seam (so that adding the offset removes the discontinuity). This reduces the problem of image stitching to the solution of a Poisson equation.

As observed by Agarwala, the offset function should only be high frequency near the seams and can be well-represented using an adaptive quadtree. We represent the target gradient field using mixed-degree finite elements stored along (dual) edges.

An example of the stitching can be seen in Figure 7. The top two rows show the input composite (with visible seams between individual images of the panorama), as well as a mask showing the assignment of pixels in the composite to input images. The bottom two rows show the results obtained using our adaptive multi-grid solver. As the figure shows, without adapted successive underrelaxation (second row from bottom), ringing artefacts are prominent. With (bottom row), these artefacts disappear. Figure 11 in the supplemental compares the stitching results obtained with different underrelaxation parameters α and β.

While it is difficult to compare directly to the method of Agarwala, we were able to compare our approach to the Distributed Multi-grid (DMG) solver [KSH10] which was shown to outperform the approach of Agarwala, both in terms of running time and memory usage. Table 2 compares the running time and memory usage of DMG with the running time and memory usage of our adaptive multi-grid solver. As the table shows, our solver provides significant

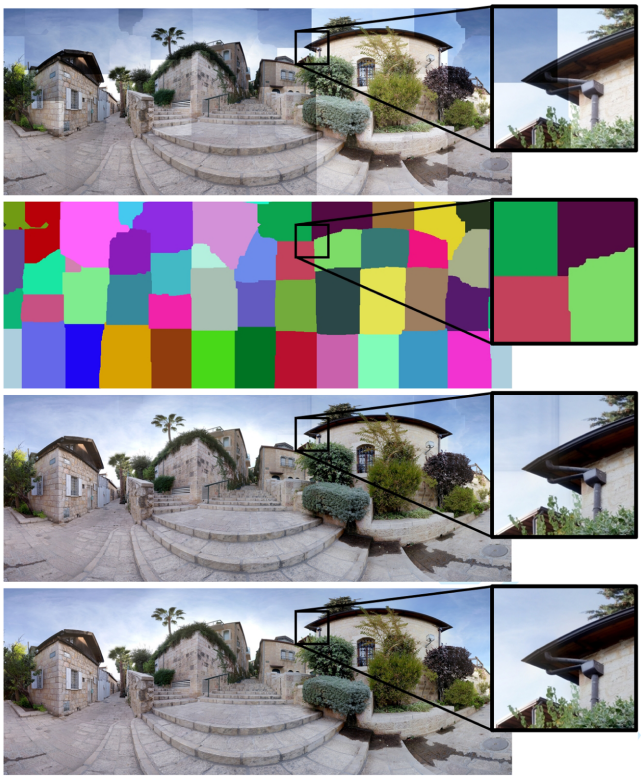

Figure 7: *Image stitching showing the composite and assignment mask for the individual images making up the Jerusalem panorama (top two rows), the results of stitching without (third row) and with (bottom row) adapted underrelaxation.*

Table 2: *Comparison of running time (seconds) and memory usage (MB) for the DMG solver [KSH10] with the implementations using our adaptive solver. (I/O times for reading/decompressing the input and writing/compressing the output are also provided.) For our implementation, the system is solved by performing a single V-cycle multi-grid pass.*

			I/O	DMG		Ours	
	Pixels	Seams	sec.	sec.	MB	sec.	MB
Emilion	9 MP	8K	1	5	119	2	13
Beynac	11 MP	5K	1	6	124	2	12
Rainier	23 MP	10K	2	6	180	4	17
PNC3	29 MP	27K	2	16	183	4	29
Sedona	34 MP	17K	4	10	192	6	21
Edinburgh	50 MP	68K	5	13	207	9	64
Crag	64 MP	36K	7	15	230	10	36
Red Rock	87 MP	50K	7	20	253	13	49
Jaffa	230 MP	197K	17	46	287	28	158
Black Tusk	240 MP	143K	18	46	286	29	118
Douthat	320 MP	187K	23	64	329	38	151
Cad Idris	380 MP	311K	28	75	339	48	245
Tallinn 2	420 MP	220K	27	77	355	47	178
Jerusalem	440 MP	277K	32	85	344	54	220
Shenandoah	648 MP	403K	53	133	378	87	324
Tallinn 7	860 MP	568K	74	165	369	118	442
Old Rag	1 GP	663K	93	180	473	142	513
Old City	3 GP	2M	234	555	527	464	1517
St. James 2	3 GP	3M	417	700	540	760	2687

performance improvements at all but the largest images. (Considering the entire time required to perform the stitching, our method runs in 60% of the time and uses 40% of the memory. Discounting the time required to read in the input pixels and labels, and to write the output, our method runs in 40% of the time.)

We believe that the deteriorating performance for high-resolution images is due to the fact that DMG has a memory footprint that is linear in the width of the image while the memory footprint of our approach is linear in the number of seams. Assuming that the resolution of individual images is constant, the number of individual images, and hence the number of seams, grows linearly with the size of the panorama. Thus, the DMG implementation has a memory footprint of $O(N^{1/2})$ while our implementation has a memory footprint of $O(N)$, where N is the number of pixels.

See Figure 10 in the supplemental for additional results.

6.3. EDT computation

In this application, the goal is to compute an approximation of the unsigned EDT of a surface in 3D (i.e. the function giving the unsigned distance from each point in 3D to the nearest point on the surface). Following the Geodesics-in-Heat approach of Crane *et al.* [CWW13], an approximation of the EDT can be computed by solving two linear systems. The first diffuses the rasterization of the surface into a 3D grid, transforming a compactly supported representation of the surface into a globally supported function. Using the fact that the gradients of the smoothed rasterization point away from the surface, the second system fits a distance function to the normalized gradients of the smoothed rasterization. Combining the two steps, one obtains a smoothed approximation of the EDT, with the extent of smoothing determined by the diffusion time in the first step.

As both systems use only first-order derivatives, Geodesics-in-Heat can be implemented using degree-one finite elements. Furthermore, because the smoothed rasterization and the output EDT are only high frequency near the input surface, they can be compactly represented using an adaptive octree.

Figure 8 (top) shows a visualization of the EDTs computed for the Stanford Bunny model, showing the exact EDT computed using the method of Saito and Toriwaki [ST94] (left) and the approximate EDT computed using our adaptive multi-grid solver without (centre) and with (right) adapted successive underrelaxation. For the visualizations, we extracted isosurfaces at regularly sampled isovalues and trimmed the isosurfaces with the yz-plane to reveal the interior surfaces. (The original surface is overlaid with transparency.) Figure 13 in the supplemental compares the EDTs obtained with different underrelaxation parameters α and β.

The advantage of our approach is highlighted in Table 3 which compares the running time and memory usage of our approximate implementation with the implementation from [ST94], at different depths. The implementation of Saito and Toriwaki has linear running

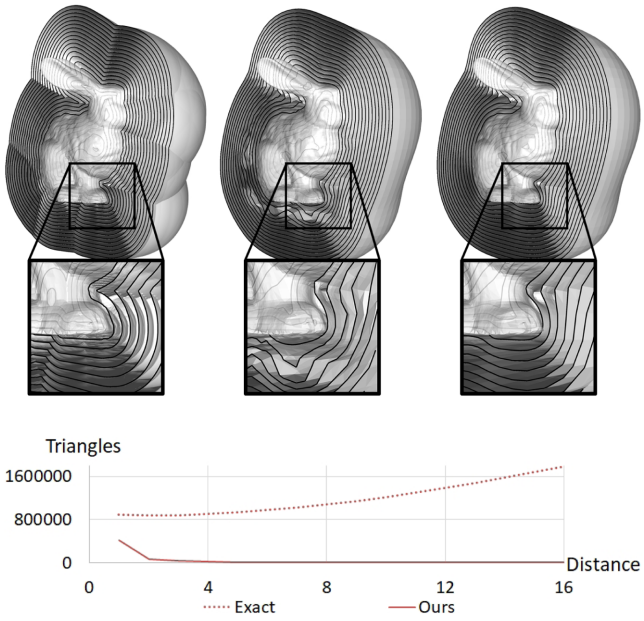

Figure 8: *(Top) Comparison of the EDT obtained for the Bunny model at resolution* 512^3 *using the approach from [ST94] (left) with the one obtained by solving the Geodesics-in-Heat system over an octree of depth 9, without (middle) and with (right) adapted underrelaxation. The figure shows regularly sampled level sets of the EDT, trimmed to the* yz*-plane. (Bottom) Output mesh complexity (in triangles) as a function of the distance from the surface.*

time and memory usage and provides an efficient solution at coarser depths/resolutions as it is trivially parallelizable and only requires a regular grid as a data structure. (At depth 8, the method of Saito and Toriwaki runs in 40% of the time, but requires 2.5 times the memory.) In contrast, our approach is only linear in the number of octree nodes and becomes more efficient at finer depths/resolutions. (At depth 9, our method runs in 85% of the time and uses 1/11th of the memory. At depth 10 our method runs in 20% of the time and uses 1/27th of the memory. At depth 11, the method of Saito and Toriwaki cannot maintain the 2048^3 voxel grid in memory.)

A further advantage of our approach is highlighted in the plot in Figure 8 (bottom), which shows the output level-set complexity as a function of the distance from the input mesh. Using a regular grid, the number of triangles grows with the surface area, and hence with distance. In contrast, using an adaptive octree, grid cells become larger away from the surface, resulting in larger marching-cubes triangles and hence lower-tessellation outputs.

We note that our approach fails to capture the fine detail of the exact solution. There are two reasons for this: (1) The Geodesics-in-Heat approach computes a smoothed approximation of the EDT, with the extent of smoothing growing as the diffusion time is increased; (2) As our octree only adapts to the input geometry, our solution is necessarily smooth away from the surface and fails to resolve the high-frequency detail near the medial axis of the shape.

See Figure 12 in the supplemental for additional results.

7. Summary and Future Work

We have described an extension of the solver initially presented in [KBH06] and [KH13] for solving the (screened) Poisson equation,

Table 3: *Comparison of running time (seconds) and memory usage (MB) for the exact Euclidean Distance Transform calculation [ST94] with the implementations obtained using our adaptive solver, at depths 8, 9, 10 and 11. For our implementation, the diffusion system is solved by performing a single V-cycle multi-grid pass while the gradient-fitting system is solved by performing a single cascadic multi-grid pass.*

			Exact		Ours	
	Area	Triangles	sec. 8/9/10/11	MB 8/9/10/11	sec. 8/9/10/11	MB 8/9/10/11
Iris	0.03	20K	1/6/86/*	132/1028/8196/*	1/2/4/13	15/28/77/252
Raptor	0.05	1716K	1/6/78/*	171/1067/8235/*	4/5/8/20	138/138/138/337
Octopus	0.10	276K	1/5/77/*	138/1034/8202/*	2/3/9/29	25/55/181/675
Fish	0.15	116K	1/5/74/*	134/1030/8198/*	1/4/13/44	28/78/272/1032
Stool	0.15	2K	0/5/81/*	131/1027/8195/*	1/3/11/41	28/75/255/940
Chair	0.16	5K	0/5/81/*	131/1027/8195/*	1/3/10/38	26/73/250/930
Nefertiti	0.19	2M	2/7/75/*	178/1074/8242/*	5/8/19/60	161/161/326/1229
Elephant	0.19	29K	1/5/78/*	132/1028/8196/*	1/4/14/55	32/96/346/1315
Hand	0.19	98K	1/6/82/*	134/1030/8198/*	1/4/14/55	35/102/354/1318
Armadillo	0.21	338K	1/6/80/*	139/1035/8203/*	2/5/17/64	35/106/379/1440
Fertility	0.21	472K	1/6/75/*	142/1039/8206/*	2/5/17/60	41/97/350/1349
Kitten	0.21	268K	1/5/68/*	138/1034/8202/*	2/5/17/64	34/106/384/1463
Dragon	0.22	851K	1/6/70/*	151/1047/8215/*	3/7/20/71	71/104/383/1489
Porsche	0.24	6K	0/5/67/*	131/1027/8195/*	1/4/14/58	28/89/336/1344
Bunny	0.30	204K	1/5/73/*	136/1032/8200/*	2/7/23/87	43/140/516/1994
Knot	0.31	391K	1/5/72/*	140/1037/8204/*	3/8/25/95	49/159/566/2166
Gargoyle	0.33	418K	1/5/72/*	141/1037/8205/*	3/8/26/99	47/157/586/2269

to a general-purpose adaptive multi-grid solver. We have demonstrated several image- and geometry-processing applications in 2D and 3D, and have shown that the abstraction of the solver does not come at the cost of either running time or memory usage.

In the future, we would like to extend our solver in several ways:

Boundary Conditions To support applications in fluid simulations, we would like to extend the solver to support boundary constraints. A simple approach would be to use the existing capacity for prescribing point-based constraints to enforce boundary conditions as a soft constraint. A better solution may be to enforce boundary constraints explicitly through integration at the finest resolution, and then using the Galerkin method to restrict to coarser resolutions.

Ringing Though our solution to the ringing problem works well in practice, we would like to consider more principled ways for resolving this phenomenon. In particular, we would like to devise a method that works consistently, independent of the number of V-cycles, the number of Gauss–Seidel iterations, the order of the B-spline or the type of linear system being solved. One approach may be to solve an inhomogeneous system where octree nodes are assigned a proximity weight which is used to modulate the contribution of the associated integral to the linear system. This would encourage the solver to focus away from the refinement boundaries in the restriction phase, reducing the overfitting at these boundaries and thereby alleviating the ringing.

Acknowledgements

We are very grateful to Aseem Agarwala and Matthew Uyttendaele for sharing their image panoramas, and to the EPFL Computer Graphics and Geometry Laboratory, the Stanford 3D Scanning Repository, Aim@Shape, Nico Schertler and Julie Digne for sharing their 3D data. We would also like to thank Fatih Calakli, Daniel Moreno and Gabriel Taubin for sharing their SSD Surface Reconstruction implementation.

This work is supported by the NSF award 1422325.

References

[Aga07] Agarwala A.: Efficient gradient-domain compositing using quadtrees. *ACM Transactions on Graphics 26* (2007), 94:1–94:5.

[Bat17] Batty C.: A cell-centered finite volume method for the Poisson problem on non-graded quadtrees with second order accurate gradients. *Journal of Computational Physics 331* (2017), 49–72.

[Bur88] Burt P. J.: Moment images, polynomial fit filters and the problem of surface interpolation. In *Proceedings of the IEEE Computer Vision and Pattern Recognition* (Ann Arbor, MI, USA, 1988), pp. 144–152.

[BXH10] Batty C., Xenos S., Houston B.: Tetrahedral embedded boundary methods for accurate and flexible adaptive fluids. *Computer Graphics Forum 29* (2010), 695–704.

[CGFO06] Chentanez, N., Goktekin, T., Feldman, B., O'Brien, J.: Simultaneous coupling of fluids and deformable bodies. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Vienna, Austria, 2006), pp. 83–89.

[CK11] Chuang M., Kazhdan M.: Interactive and anisotropic geometry processing using the screened Poisson equation. *ACM Transactions on Graphics 30* (2011), 57:1–57:10.

[CM11] Chentanez N., Müller M.: Real-time Eulerian water simulation using a restricted tall cell grid. *ACM Transactions on Graphics 30* (2011), 82:1–82:10.

[CMT11] Calakli, F., Moreno, D., Taubin, G.: SSD surface reconstruction (v 3.0). http://mesh.brown.edu/ssd/software.html (Retrieved November 30, 2017).

[CT11] Calakli F., Taubin G.: SSD: Smooth signed distance surface reconstruction. *Computer Graphics Forum 30* (2011), 1993–2002.

[CT12] Calakli, F., Taubin, G.: *SSD-C: Smooth Signed Distance Colored Surface Reconstruction*. Springer London, London, 2012, pp. 323–338.

[CWW13] Crane K., Weischedel C., Wardetzky M.: Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Transactions on Graphics 32* (2013), 152:1–152:11.

[DHK*00] Douglas C., Hu J., Kowarschik M., Rüde U., Weiss C.: Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis 10* (2000), 21–40.

[FKG15] Fuhrmann, S., Kazhdan, M., Goesele, M.: Accurate isosurface interpolation with Hermite data. In *3DV '15: Proceedings of the 2015 International Conference on 3D Vision* (Lyon, France, 2015), pp. 256–263.

[FOK05] Feldman B., O'Brien J., Klingner B.: Animating gases with hybrid meshes. *ACM Transactions on Graphics 24* (2005), 904–909.

[FWD14] Ferstl F., Westermann R., Dick C.: Large-scale liquid simulation on adaptive hexahedral grids. *IEEE Transactions on Visualization and Computer Graphics 20* (2014), 1405–1417.

[GGSC96] Gortler, S., Grzeszczuk, R., Szeliski, R., Cohen, M.: The lumigraph. In *Proceedings of Computer Graphics and Interactive Techniques* (New Orleans, LA, USA, 1996), pp. 43–54.

[HW65] Harlow F., Welch J.: Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids 8* (1965), 2182–2189.

[IGLF06] Irving G., Guendelman E., Losasso F., Fedkiw R.: Efficient simulation of large bodies of water by coupling two and three dimensional techniques. *ACM Transactions on Graphics 25* (2006), 805–811.

[Kaz13] KAZHDAN, M.: Screened Poisson Surface Reconstruction (v 5.71). http://www.cs.jhu.edu/misha/Code/PoissonRecon/ (Retrieved November 30, 2017).

[KBH06] KAZHDAN, M., BOLITHO, M., HOPPE, H.: Poisson surface reconstruction. In *Proceedings of Symposium on Geometry Processing* (Cagliari, Sardinia, Italy, 2006), pp. 61–70.

[KFCO06] KLINGNER B., FELDMAN B., CHENTANEZ N., O'BRIEN J.: Fluid animation with dynamic meshes. *ACM Transactions on Graphics 25* (2006), 820–825.

[KH13] KAZHDAN M., HOPPE H.: Screened Poisson surface reconstruction. *ACM Transactions on Graphics 32* (2013), 29:1–29:13.

[KM90] KASS M., MILLER G.: Rapid, stable fluid dynamics for computer graphics. *ACM SIGGRAPH Computer Graphics 24* (1990), 49–57.

[KSH10] KAZHDAN M., SURENDRAN D., HOPPE H.: Distributed gradient-domain processing of planar and spherical images. *ACM Transactions on Graphics 29* (2010), 14:1–14:11.

[LGF04] LOSASSO F., GIBOU F., FEDKIW R.: Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics 23* (2004), 457–462.

[MGD*10] MULLEN P., GOES F. D., DESBRUN M., COHEN-STEINER D., ALLIEZ P.: Signing the unsigned: Robust surface reconstruction from raw pointsets. *Computer Graphics Forum 29* (2010), 1733–1741.

[OH95] O'BRIEN, J., HODGINS, J.: Dynamic simulation of splashing fluids. In *Proceedings of the Computer Animation* (Geneva, Switzerland, 1995), pp. 198–205.

[Pfe63] PFEIFER C.: Data flow and storage allocation for the PDQ-5 program on the Philco-2000. *Communications of the ACM 6* (1963), 365–366.

[Pop03] POPINET S.: Gerris: A tree-based adaptive solver for the incompressible Euler equations in complex geometries. *Journal of Computational Physics 190* (2003), 572–600.

[SABS14] SETALURI R., AANJANEYA M., BAUER S., SIFAKIS E.: SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics 33* (2014), 205:1–205:12.

[Sam90] SAMET H.: *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., Reading, MA, 1990.

[ST94] SAITO T., TORIWAKI J.: New algorithms for Euclidean distance transformation of an n-dimensional digitized picture with applications. *Pattern Recognition 27* (1994), 1551–1565.

[TRS06] THÜREY, N., RÜDE, U., STAMMINGER, M.: Animation of open water phenomena with coupled shallow water and free surface simulations. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Vienna, Austria, 2006), pp. 157–164.

[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* (San Diego, CA, USA, 2007), pp. 229–238.

[WS93] WARREN, M., SALMON, J.: A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (Portland, OR, USA, 1993), pp. 12–21.

Supporting Information

Additional supporting information may be found online in the Supporting Information section at the end of the article.

Figure 9: Surface reconstructions, showing (from left to right) the results of the Screened Poisson Reconstruction (degree 1), Screened Poisson Reconstruction (degree 2), the original implementation of the Smoothed Signed Distance Reconstruction by Calakli et al., and our implementation of the Smoothed Signed Distance Reconstruction.

Figure 10: Image stitching results, showing the input composite and assignment mask making up the panorama (left), and the output (right).

Figure 11: Close-ups from the Jerusalem panorama stitching results for different under-relaxation parameters α and β. (See Figure 7 in the main text for the full panorama.) Note that the bottom-left close-up corresponds to the result without under-relaxation and the close-up highlighted in red corresponds to the result with our default parameters.

Figure 12: Isocontours of the Euclidean distance transform estimated using the method of Saito and Toriwaki (left) compared to the isosurfaces obtained by solving the Geodesics-In-Heat problem over an adaptive octree (right).

Figure 13: Close-ups from the isocontours of the Euclidean distance transform obtained from the bunny model for different under-relaxation parameters α and β.