The Visual Computer (2024) 40:2081–2088
https://doi.org/10.1007/s00371-023-02903-0

ORIGINAL ARTICLE

The inverse barycentric displacement problem

Andrea Maggiordomo[1] · Yury Uralsky[2] · Henry Moreton[2] · Marco Tarini[1]

Accepted: 10 May 2023 / Published online: 16 June 2023

Abstract
In this short paper, we analyze the problem of finding the triangular barycentric coordinates of an interpolated ray hitting a given point. This task, which we term the *inverse barycentric displacement problem*, is general and useful in geometry processing and computer graphics. Concrete applications of the solution include the construction of displacement maps and texture baking. We derive the set of complete, closed-form solutions and discuss the number and existence of solutions. We close with a discussion of implementation-oriented optimizations and a few example applications.

Keywords Displacement mapping · Vertex projection · Inverse ray casting · Texture baking

1 Introduction

Consider a 3D embedded triangle $\mathcal{T}$ with arbitrary displacement vectors ($\mathbf{d}_i$) defined at its corners ($\mathbf{v}_i$) (Fig. 1, *left*). A ray is defined starting at a barycentrically interpolated point $\mathbf{v}$ on $\mathcal{T}$, where $\mathbf{v} = u\mathbf{v}_0 + v\mathbf{v}_1 + w\mathbf{v}_2$, and traveling in the direction of the (corresponding) barycentrically interpolated displacement $\mathbf{d}$. Traveling along this ray, a displaced point $\mathbf{q}$ is defined at parametric position t:

$$\mathbf{q} = \mathbf{v} + t\,\mathbf{d} \tag{1}$$

t is the *scalar displacement value* defined at $\mathbf{v}$ and may be stored in a scalar displacement texture covering $\mathcal{T}$. In a common scenario, we are given an interpolated position $\mathbf{v}$ and direction $\mathbf{d}$ on $\mathcal{T}$, and we want to find $\mathbf{q}$, typically on a given *target* surface (dotted line in Fig. 1, *right*). For example, $\mathbf{q}$ can be found by ray casting against a high-resolution mesh. Here, we use the term *direct* displacement to refer to the problem of finding $\mathbf{q}$ (equivalently, t) for specified $\mathbf{v}$ and $\mathbf{d}$.

In this work, we are interested in the *inverse* of the *direct* displacement problem, where we are given a target point $\mathbf{q}$ and we want to find a position $\mathbf{v}$ on $\mathcal{T}$ such that the ray defined on $\mathbf{v}$ hits $\mathbf{q}$. The solution is interesting, useful, and non-trivial.

Rationale: The inverse problem is useful in several contexts. For example, it makes it possible to bake or compute a displacement map, or a texture for $\mathcal{T}$ by processing positions $\mathbf{q}$ directly on the target displaced surface, this sidesteps the need for ray casting, which typically requires expensive iteration over the numerous elements of a high-resolution, tessellated target surface. The design of good support meshes to be displaced can also benefit from the ability to quickly identify where on a potential $\mathcal{T}$, if anywhere, a given point $\mathbf{q}$ on the target surface would be mapped.

The inverse mapping problem is not trivial, and, in spite of its potential usefulness, has never been solved in closed form (to our knowledge). The nontriviality is confirmed by the observation that the solution does not necessarily exist, nor is it necessarily unique, and certain configurations, for example when all three rays meet into $\mathbf{q}$, even admit infinite solutions for $\mathbf{v}$, as $\mathbf{q}$ can be reached from any point on $\mathcal{T}$.

Related work: Traditionally, in contexts such as texture baking, displacement-map construction (for curved surface representations or polygonal meshes), inverse subdivision surfaces, and others, existing solutions are designed to leverage solely on the solution of the direct problem. This may be due, at least in part, to the perceived intricacy of the inverse problem, which is the subject of this work. In Sect. 5, we exemplify two scenarios where our closed-form solution to the inverse problem offers a compelling alternative to traditional solutions. The description of each scenario starts with a brief overview of the respective Related Work.

✉ Marco Tarini
marco.tarini@unimi.it

1 Università degli Studi di Milano, Milan, Italy

2 NVIDIA, Santa Clara, CA, USA

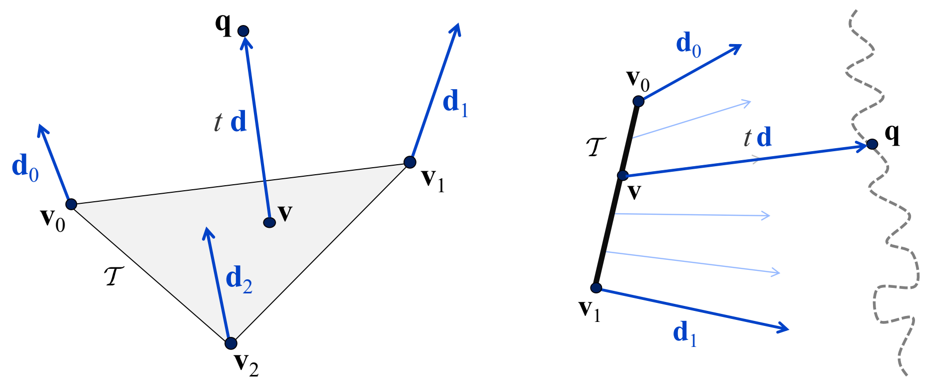


Fig. 1 *left* - The inputs of the displacement problem (direct or inverse.) *right* - A 2D depiction (note that in 3D, the 3 corner rays do not meet at a point, which would otherwise make the problem trivial.)

2 Problem definition

Given a target point $\mathbf{q}$, we want to find a point $\mathbf{v}$ on $\mathcal{T}$ such that the ray shot from $\mathbf{v}$ in the interpolated direction $\mathbf{d}$ hits $\mathbf{q}$ (1).

Let the corners of $\mathcal{T}$ be in positions $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$, and let the three directions defined at corners be $\mathbf{d}_0$, $\mathbf{d}_1$, $\mathbf{d}_2$. A point $\mathbf{v}$ is identified by its nonnegative barycentric coordinates $\alpha_0, \alpha_1, \alpha_2$ with

$$\sum_i \alpha_i = 1 \tag{2}$$

so that:

$$\mathbf{v} = \sum_i \alpha_i \, \mathbf{v}_i \tag{3}$$

$$\mathbf{d} = \sum_i \alpha_i \, \mathbf{d}_i \, . \tag{4}$$

Substituting (3, 4) in (1), gives

$$\mathbf{q} = \sum_i \alpha_i \, \mathbf{v}_i + t \sum_i \alpha_i \, \mathbf{d}_i \tag{5}$$

where the three scalars α_i and the scalar t are unknowns.

This seemingly simple geometric setup leads to a system, (5), of three scalar equations, quadratic with the four variables, subject to the linear constraint (2). This system is not trivial to solve directly.

3 Geometric derivation of the solution

We can rewrite Equation (5) as

$$\begin{aligned}\mathbf{q} &= \sum_i \alpha_i \, (\mathbf{v}_i + t \, \mathbf{d}_i) \\ &= \sum_i \alpha_i \, \mathbf{a}_i \end{aligned}\tag{6}$$

with $\mathbf{a}_{0,1,2}$ denoting the position of the vertices of $\mathcal{T}$ displaced by t along the respective direction:

$$\mathbf{a}_i = \mathbf{v}_i + t \, \mathbf{d}_i \tag{7}$$

Equation (6) implies that the target point $\mathbf{q}$ must necessarily lie on the plane $\mathcal{P}$ passing by the three positions $\mathbf{a}_i$ (see Fig. 1), irrespective of the choice of $\mathbf{v}$; observe that $\mathcal{P}$ is a function solely of t (and different choices of t result in general in planes with different orientations). Moreover, it implies that the barycentric coordinates of $\mathbf{q}$ in triangle $\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2$ are the same as the sought barycentric coordinates $\alpha_0, \alpha_1, \alpha_2$ of $\mathbf{v}$ in $\mathcal{T}$.

Our solution is therefore to first find value(s) of t so that $\mathbf{q}$ lies on $\mathcal{P}$, and then simply find $\alpha_0, \alpha_1, \alpha_2$ (and thus $\mathbf{v}$) with fixed t as the barycentric coordinates of $\mathbf{q}$ inside triangle $\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2$.

The condition for $\mathbf{q}$ to be on $\mathcal{P}$ can be expressed as

$$0 = (\mathbf{a}_0 - \mathbf{q}) \cdot \mathbf{n}, \tag{8}$$

that is, that the vector connecting $\mathbf{q}$ to $\mathbf{a}_0$ is orthogonal to $\mathbf{n}$, with $\mathbf{n}$ denoting a vector orthogonal to $\mathcal{P}$ ($\mathbf{n}$ is, again, as a function of t):

$$\mathbf{n} = (\mathbf{a}_1 - \mathbf{a}_0) \times (\mathbf{a}_2 - \mathbf{a}_0). \tag{9}$$

Rewriting (8) in terms of $\mathbf{v}$, $\mathbf{d}$ and t gives:

$$\begin{aligned} 0 &= (\mathbf{a}_0 - \mathbf{q}) \cdot (\mathbf{a}_1 - \mathbf{a}_0) \times (\mathbf{a}_2 - \mathbf{a}_0) \\ &= (\mathbf{v}_0 + t \, \mathbf{d}_0 - \mathbf{q}) \\ &\quad \cdot (\mathbf{v}_1 + t \, \mathbf{d}_1 - \mathbf{v}_0 - t \, \mathbf{d}_0) \\ &\quad \times (\mathbf{v}_2 + t \, \mathbf{d}_2 - \mathbf{v}_0 - t \, \mathbf{d}_0) \\ &= ((\mathbf{v}_0 - \mathbf{q}) + t \, \mathbf{d}_0) \\ &\quad \cdot ((\mathbf{v}_1 - \mathbf{v}_0) + t \, (\mathbf{d}_1 - \mathbf{d}_0)) \\ &\quad \times ((\mathbf{v}_2 - \mathbf{v}_0) + t \, (\mathbf{d}_2 - \mathbf{d}_0)) \end{aligned} \tag{10}$$

This gives a polynomial that is cubic in t.

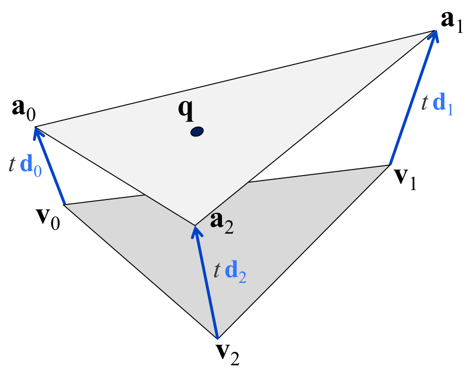


Fig. 2 Construction for the geometric solution

3.1 Deriving the polynomial coefficients

To ease the derivation of the coefficients of the polynomial in (10), it is convenient to group the constant sub-expressions in (10) as columns of two 3×3 matrices:

$$\mathbf{P} = \left[\, \mathbf{v}_0 - \mathbf{q} \,|\, \mathbf{v}_1 - \mathbf{v}_0 \,|\, \mathbf{v}_2 - \mathbf{v}_0 \,\right], \tag{11}$$
$$\mathbf{D} = [\quad \mathbf{d}_0 \quad |\, \mathbf{d}_1 - \mathbf{d}_0 \,|\, \mathbf{d}_2 - \mathbf{d}_0 \,] \tag{12}$$

We term $\mathbf{P}$ the *position matrix*, as it depends on the positions of vertices of $\mathcal{T}$ and of target point $\mathbf{q}$; we term $\mathbf{D}$ the *displacement matrix*, as it depends on the displacement directions defined at the vertices of $\mathcal{T}$. We will refer to their columns as $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$ (and likewise for $\mathbf{D}$).

We can now rewrite (10) as

$$0 = (\mathbf{P}_0 + t\,\mathbf{D}_0) \cdot (\mathbf{P}_1 + t\,\mathbf{D}_1) \times (\mathbf{P}_2 + t\,\mathbf{D}_2) \tag{13}$$

That is (remembering that $\det(\mathbf{a}|\mathbf{b}|\mathbf{c})$ is $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$):

$$0 = \det(\mathbf{P} + t\,\mathbf{D})\,. \tag{14}$$

Extracting t gives (see Appendix for a derivation):

$$0 = A\,t^3 + B\,t^2 + C\,t + D \tag{15}$$

with

$$\begin{aligned}
A &= \det(\mathbf{D})\\
B &= \det(\mathbf{P}_0|\mathbf{D}_1|\mathbf{D}_2) + \det(\mathbf{D}_0|\mathbf{P}_1|\mathbf{D}_2) + \det(\mathbf{D}_0|\mathbf{D}_1|\mathbf{P}_2)\\
C &= \det(\mathbf{D}_0|\mathbf{P}_1|\mathbf{P}_2) + \det(\mathbf{P}_0|\mathbf{D}_1|\mathbf{P}_2) + \det(\mathbf{P}_0|\mathbf{P}_1|\mathbf{D}_2)\\
D &= \det(\mathbf{P})
\end{aligned} \tag{16}$$

3.2 Finding all valid solutions (if any)

Solving (15) with the standard cubic formula returns up to three real solutions for t, which we denote as t_0, t_1, and t_2.

For a given choice of t_i, the corresponding triangle $\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2$ is constructed (7), then, α_i are trivially extracted as the barycentric coordinates of $\mathbf{q}$ in that triangle:

$$\alpha_i = \frac{\mathbf{n} \cdot (\mathbf{q} - \mathbf{a}_{i+2}) \times (\mathbf{a}_{i+1} - \mathbf{a}_{i+2})}{\mathbf{n} \cdot \mathbf{n}}. \tag{17}$$

If any $\alpha_i < 0$, then $\mathbf{v}$ is outside the triangle $\mathcal{T}$, and the corresponding solution t_i must be discarded (after [14], this can be conveniently tested prior to the division by $\mathbf{n} \cdot \mathbf{n}$).

This procedure fails when the triangle $\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2$ is degenerate, that is, when it collapses into a line or a point (meaning that the used value of t_i satisfies (8) but only in the trivial sense that $\mathbf{n}$ is the zero vector). This condition can be detected by checking that $\mathbf{n}$ (9) vanishes. If the target point $\mathbf{q}$ happens to belong to this line or point, then we have infinite solutions for α_i – and therefore, with (4), for $\mathbf{v}$; otherwise, the solution t_i must be discarded.

Depending on the context, negative values of t_i, or values larger than one, may also be discarded as invalid (when "backward" negative displacement, or displacements shooting "beyond $\mathbf{d}$," are to be disallowed).

Each non-discarded t_i, if any, results in a valid solution to the inverse projection problem. Multiple such solutions can exist. A possible strategy to employ in such cases is to consistently favor the smallest value of $|t_i|$, which has the benefit of making $\mathbf{v}$ vary with continuity as $\mathbf{q}$ varies (barring degenerate cases).

Special cases: It is interesting to analyze a few special cases of the general formulation above, although an implementation does not need to be aware of them.

When $\mathbf{q}$ is on $\mathcal{T}$, or on the plane containing $\mathcal{T}$, then the position matrix $\mathbf{P}$ is singular (its three columns are co-planar), so D is zero, and $t = 0$ is always a solution of (15), as expected (observe that it is not necessarily the only solution).

When the displacement vectors $\mathbf{d}_i$ are all parallel to each other (even if with different lengths), then the direction matrix $\mathbf{D}$ is singular and rank 1 (as the determinant and rank of $\mathbf{D}$ are the same as $[\mathbf{d}_0|\mathbf{d}_1|\mathbf{d}_2]$). This makes both A and B zero (as taking any two vectors from $\mathbf{D}$ zeroes the determinant), making the polynomial (15) linear, and the solution for t unique. This is also expected, from a geometric point of view (observe however that this still does not make all planes $\mathcal{P}$ parallel to each other, for all t: that additionally requires all displacement vectors to be the same length).

A pathological case is when all $\mathbf{d}_i$ are in the same plane as $\mathcal{T}$. In that case, A, B, and C are zeroed (as each matrix is made by co-planar column vectors), making the polynomial (15) degree 0. Unless D is also 0, when $\mathbf{q}$ is also on the plane of $\mathcal{T}$, there is no solution for t, valid or otherwise, as expected.

4 Implementation and optimizations

A reference implementation (in C++) is provided in the additional materials.

Workload: For the sake of simplicity, we will estimate the workload by counting the number of floating-point products (disregarding sums; among other reasons, because most can be absorbed by Multiply-and-Add operations). Finding the coefficients as in (**??**,16) involves the computations of 8 determinants, that is, 8 cross products and 8 dot products, amounting to 72 products (each cross costing 6 floating-point products, and each dot costing 3).

Common sub-expressions optimization: using three temporary vectors, $\mathbf{t_{DD}}$, $\mathbf{t_{PD}}$ and $\mathbf{t_{PP}}$, allows the computation of A, B, C, D with only 4 cross products and 6 dot products (42 products in total):

$$\begin{aligned}
\mathbf{t_{DD}} &\leftarrow \mathbf{D}_1 \times \mathbf{D}_2\\
\mathbf{t_{PD}} &\leftarrow \mathbf{P}_1 \times \mathbf{D}_2 + \mathbf{D}_1 \times \mathbf{P}_2\\
\mathbf{t_{PP}} &\leftarrow \mathbf{P}_1 \times \mathbf{P}_2
\end{aligned}$$

(the latter is also the area vector of $\mathcal{T}$) then rewriting (16) as

$$\begin{aligned}
A &\leftarrow \mathbf{D}_0 \cdot \mathbf{t_{DD}} & D &\leftarrow \mathbf{P}_0 \cdot \mathbf{t_{PP}}\\
B &\leftarrow \mathbf{P}_0 \cdot \mathbf{t_{DD}} + \mathbf{D}_0 \cdot \mathbf{t_{PD}} & C &\leftarrow \mathbf{D}_0 \cdot \mathbf{t_{PP}} + \mathbf{P}_0 \cdot \mathbf{t_{PD}}
\end{aligned}$$

Caching optimization: When the inverse projection problem must be solved for multiple target points $\mathbf{q}$ over the same triangle $\mathcal{T}$, only $\mathbf{P}_0$ varies in each instance of the problem: The three temporary vectors, as well as three scalars A, $\mathbf{D}_0 \cdot \mathbf{t_{PD}}$, and $\mathbf{D}_0 \cdot \mathbf{t_{PP}}$, can be cached for a given triangle, reducing the cost of finding the four coefficients for a new $\mathbf{q}$ to just three dot products (9 products in total).

5 Example applications

As we argued in Sect. 1, the presented inverse problem can find applications in several contexts in Geometry Processing. In the following, we briefly exemplify this with two use cases. A full implementation of both examples is attached in the additional material.

5.1 Texture baking via rendering

Texture baking is a common technique, pioneered by early research such as [3, 9] and supported today by common 3D modeling suites [1, 5, 7, 12]. A baked texture (sometimes referred to as transfer map, mesh map, redetail texture, or detail-recover texture) is a synthesized texture that records data originally stored in a high-resolution mesh $\mathcal{M}_H$ into the texels of a texture image intended for a given low-resolution mesh $\mathcal{M}_L$. The texels can be filled with displacement scalars, or any information stored on $\mathcal{M}_H$, such as normals, colors, pre-lit values, or any other per-vertex attributes, per face attributes, or even an original texture; $\mathcal{M}_L$ comes with a UV-map, which for simplicity we will assume to be free from seams, and per-vertex displacement directions; the latter can be per-vertex normal directions, or can even be explicitly optimized for this specific task in preprocessing [15].

Direct approach: Traditionally, the baking is performed leveraging on the *direct* problem: For each texel, a ray is cast from the corresponding position on $\mathcal{M}_L$, toward the interpolated direction, and the information is extracted and stored from whichever point on $\mathcal{M}_H$ is hit.

Inverse approach (new): Using the inverse problem, we can iterate over vertices of $\mathcal{M}_H$, find a position on $\mathcal{M}_L$ by inverse projection, and reposition that vertex in the corresponding location in the 2D texture space. This way, $\mathcal{M}_H$ is morphed into the 2D parametric space of $\mathcal{M}_L$, and a simple rasterization-based rendering suffices to produce the baked texture. Alternatively, depth values t can also be rendered to bake displacement maps. The attached implementation demonstrates this process. See Fig. 3 for one example.

Vertices of $\mathcal{M}_H$ that fail to be back-projected over any face of $\mathcal{M}_L$ with a valid solution can either simply be discarded, or clamped to the boundary of the closest triangle in $\mathcal{M}_L$ (zeroing the least negative barycentric coordinate and re-normalizing the other two).

Comparison: The relative merits of the two solutions, traditional (direct) and novel (inverse), depend on the context, and a full analysis exceeds the scope of this paper; for our intents, the observation suffices that the two solutions offer a different set of advantages. In the following, we outline only a few general considerations.

Efficiency can be compared, neglecting for the sake of simplicity the impact of spatial indexing structures. A clear advantage of the inverse approach is that the final rendering pass can exploit the extremely optimized standard rasterization rendering pipeline; this is in practice extremely fast (milliseconds at most), and we can safely disregard its impact on computation time. Let R_H and R_L be the resolutions of $\mathcal{M}_H$ and $\mathcal{M}_L$ (number of primitives, either vertices or faces), and R_T be the resolution of the baked texture (number of texels), with, typically

$$R_T > R_H \gg R_L \tag{18}$$

(to prevent loss of information, R_T needs to surpass R_H, e.g. by a factor 2 according to Nyquist-Shannon sampling theorem; considerably more if $\mathcal{M}_H$ comes with its own texture to be re-baked). For a given triangle, both the inverse and direct problems are solved in constant time, although direct ray

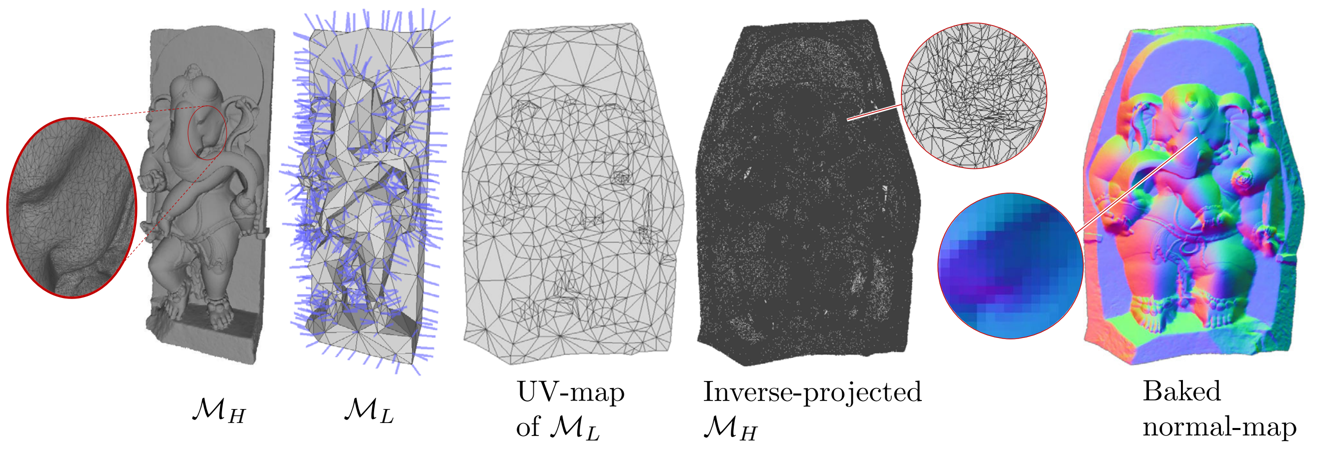


Fig. 3 Texture baking via inverse displacement: an example of a result. The input consists of a hi-resolution mesh $\mathcal{M}_H$ (324K faces) and a low-resolution mesh $\mathcal{M}_L$ (1000 faces), provided with per-vertex displacement directions (in blue), and a UV-map (shown). We inverse project every vertex of $\mathcal{M}_H$ into $\mathcal{M}_L$, moving it into the corresponding u, v position, and rasterize it using a normal rendering pass, obtaining the baked normal-map. *Used input data:* $\mathcal{M}_H$ is the *Ganesa* model from [11]; $\mathcal{M}_L$ is obtained by automatic simplification, using MeshLab; displacement directions on $\mathcal{M}_L$ are standard per-vertex normal directions (area-weighted average of per-face normal); the UV-map of $\mathcal{M}_L$ is constructed with [10]

casting necessitates fewer operations. In the direct method, a ray from each texel must be tested against each primitive in $\mathcal{M}_H$, leading to a $O(R_T R_H)$ complexity; in the inverse approach, all vertices of $\mathcal{M}_H$ must be inverse-projected over each triangle of $\mathcal{M}_L$, costing $O(R_H R_L)$, which is a much lower asymptotic complexity. Naturally, both approaches can be sped up considerably by employing spatial indexing structures (down to $O(R_T \log(R_H))$ and $O(R_H \log(R_L))$, respectively), which reduces the gap. Even without that non-trivial optimization, our reference implementation takes only a few seconds on consumer-level hardware for the examples in Fig. 3.

Ease of use favors the inverse-projection approach, as fragment-shaders, e.g. the same ones used in the rendering, can be directly re-used in the final pass to reproduce any complex lighting or texturing effect to bake the results in form of textures (including shadowing, wireframe, or any other mesh rendering technique).

Accuracy favors the direct approach, as in the inverse approach the linear interpolation inside the triangles of the flattened $\mathcal{M}_H$ introduces a small approximation error, which decreases with the resolution of $\mathcal{M}_H$. Also, the inverse approach requires care to work in presence of texture seams over $\mathcal{M}_L$.

5.2 Perfect reprojectability test

In the context of the construction of a displacement-mapped surface, a recurrent concern is whether or not a given base mesh, with an associated set of per-vertex displacement directions, is able to represent a given continuous target surface, often discretized as a high-resolution mesh $\mathbf{M}_H$ [4, 8, 13]. A (scalar) displacement-map is limited in that it can only encode different heights for each point; in other words, it is a warped "2.5D" height-field, and, as such, it can only express a limited range of shapes. It is in general not trivial to determine if a given target surface $\mathbf{M}_H$ can be represented by a displacement-map encoded over a base triangle $\mathcal{T}$.

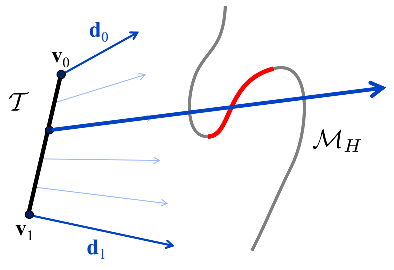


Fig. 4 2D depiction of a case where a given surface $\mathcal{M}_H$ is *not* reprojectable from base triangle $\mathcal{T}$, as one interpolated ray intersects $\mathcal{M}_H$ three times, including with one with mismatching orientation (red bold line)

Direct approach: A brute force solution using the direct approach is to cast rays from a sampling of $\mathcal{T}$ and check whether or not any given ray hits multiple valid targets on $\mathcal{M}_H$, or, equivalently, whether or not it hits any back-facing face on $\mathcal{M}_H$ (see Fig. 4). This relies on the sampling to be sufficiently dense, and is work intensive, making it impractical for example in contexts where the test must be performed to inform the construction of an arrangement of $\mathcal{T}$.

Existing alternatives: In [8], the displacement is redefined as a continuous piecewise linear mapping, to make it easier to

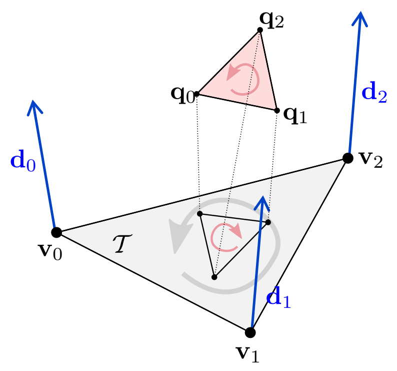


Fig. 5 Exact reprojectability test for a triangle $(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2)$, over a base triangle $\mathcal{T}$. In this instance, the test fails because the projection of $(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2)$ of $\mathcal{T}$ has a different orientation than $\mathcal{T}$

invert, making it possible to precisely test for reprojectability; while the test is exact, that framework changes the definition of the displacement map, drifting away from the traditionally intended semantics (vertices are displaced not along a single straight line, but along a broken line). In many previous works, the test is only approximated, so to make it faster. For example, in [13], the test is substituted by using a "fitmap," which is a scalar field defined in preprocessing that describes, for each position p on $\mathbf{M}_H$, the approximate maximal size of a base-triangle centered around p that can still be correctly reprojected; while this is can be an efficient way to inform the construction of an arrangement of base triangles, the test is only an approximation.

Inverse approach (new): to perfectly test for reprojectability, we can test if any triangle in $\mathcal{M}_H$ is mapped, by the inverse projection, into a triangle inside $\mathcal{T}$ having an orientation opposite to that of $\mathcal{T}$. Given a triangle $(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2) \in \mathcal{M}_H$, we first inverse-project $\mathbf{q}_0$, $\mathbf{q}_1$ and $\mathbf{q}_2$, obtain three sets of barycentric coordinates $\boldsymbol{\alpha}^0$, $\boldsymbol{\alpha}^1$ and $\boldsymbol{\alpha}^2$ on $\mathcal{T}$, with $\boldsymbol{\alpha}^i = (\alpha^i_0, \alpha^i_1, \alpha^i_2)$ – see Fig. 5.

The consistency of orientation of the back-projected triangle on $\mathcal{T}$ is then determined by the sign the 2D cross product of the 2D triangle defined by any two barycentric coordinates:

$$(\alpha^1_0 - \alpha^0_0)(\alpha^1_1 - \alpha^0_1) - (\alpha^2_0 - \alpha^0_0)(\alpha^2_1 - \alpha^0_1) > 0 \tag{19}$$

This test is exact; note that it does not explicitly require the 3D normal of either $\mathcal{T}$ or $(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2)$.

6 Conclusions

In this work, we offer a complete solution to what we term the "inverse displacement problem," a basic geometric sub-problem that commonly arises in contexts such as the construction or baking of scalar displacement-maps over meshes, subdivision surfaces, and other representations. Our solution is given in closed form, and can be made efficient to evaluate; while typical problem instances admit a single valid solution, our procedure also correctly identifies cases admitting no solution, infinite solutions, or a set of either two or three solutions to choose from.

We observe that the vast and long-spanning Literature on displacement-map construction, and adjacent fields, always strive to formulate their algorithms to employ the direct problem only (to the best of our knowledge). We conjecture this may be due, in some part, to the perceived intricacy of the inverse problem, which, in spite of its apparent geometric simplicity, defies at first glance straightforward solutions. The present work intends to offer an alternative and settle the question.

6.1 Problem variants

We conclude by considering the effect of a few common variations of the direct problem on the inverse problem.

In some scenario, the displacement vectors $\mathbf{d}_i$ are assumed to be of the same length, or all unitary. Clearly, our solution makes no assumption about this and is directly usable. As far as we can tell, this additional assumption does not offer any opportunity to ease the inverse problem.

In some scenario, the interpolated displacement vector $\mathbf{d}$ is re-normalized after interpolation, defining the ray direction as $\mathbf{d}' = \mathbf{d}/||\mathbf{d}||$. This change does not affect our construction at all, provided that our final scalar displacement value t is adjusted, after the computation is over, to $t' = t\,||\mathbf{d}||$.

A different scenario is when the base triangle $\mathcal{T}$ is substituted by a (flat or not flat) quadrilateral $\mathcal{Q}$. For example, in many techniques, such as [2, 6, 13] a displacement map is applied over a quad-mesh rather than a triangle-mesh. The direct displacement problem is not affected much, except that $\mathbf{v}$ and $\mathbf{d}$ are now bilinearly interpolated between four corners, rather than being linearly interpolated between three. The inverse problem, however, becomes considerably more intricate. The presented solution is now unusable: Equations from 2 to 7 can be immediately adapted (index i now ranging from 0 to 3), but neither $\mathcal{P}$ or $\mathbf{n}$ are defined any longer (as the four $\mathbf{a}_i$ are not necessarily co-planar, not even when $\mathcal{Q}$ is flat and $\mathbf{d}_i$ are unitary). In this variant, the inverse problem is (to our knowledge) still open.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s00371-023-02903-0.

Funding Open access funding provided by Università degli Studi di Milano within the CRUI-CARE Agreement.

Data Availability All data generated and analyzed during this study are included in this published article and its supplementary information files, including the source files.

Declarations

Conflict of interest The authors declare no competing interests.

A Derivation of (15) from (14)

For any pair of 3 × 3 matrices $\mathbf{M} = [\mathbf{M}_0|\mathbf{M}_1|\mathbf{M}_2]$ and $\mathbf{N} = [\mathbf{N}_0|\mathbf{N}_1|\mathbf{N}_2]$, we have

$$\begin{aligned}\det(\mathbf{M}+\mathbf{N}) &= \det(\mathbf{M}_0|\mathbf{M}_1|\mathbf{M}_2) + \det(\mathbf{M}_0|\mathbf{M}_1|\mathbf{N}_2)\\ &+ \det(\mathbf{M}_0|\mathbf{N}_1|\mathbf{M}_2) + \det(\mathbf{M}_0|\mathbf{N}_1|\mathbf{N}_2)\\ &+ \det(\mathbf{N}_0|\mathbf{M}_1|\mathbf{M}_2) + \det(\mathbf{N}_0|\mathbf{M}_1|\mathbf{N}_2)\\ &+ \det(\mathbf{N}_0|\mathbf{N}_1|\mathbf{M}_2) + \det(\mathbf{N}_0|\mathbf{N}_1|\mathbf{N}_2).\end{aligned} \tag{20}$$

This is an application of the general rule according to which the determinant of a sum of two $n \times n$ matrices is given by the sum of the 2^n determinants of the matrices obtained by any swapping of columns between the first and the second matrix; for the $n = 3$ case, this can also be immediately verified by rewriting $\det(\mathbf{a}|\mathbf{b}|\mathbf{c})$ as $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$.

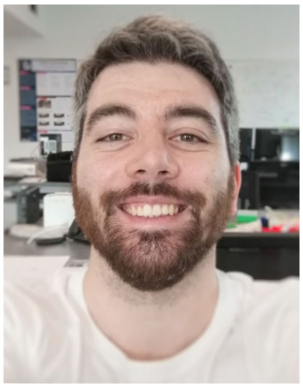
Applying (20) to (14) with $\mathbf{M} = \mathbf{P}$ and $\mathbf{N} = t\,\mathbf{D}$, and remembering that, for any scalar k,

$$\det(k\,\mathbf{a}|\mathbf{b}|\mathbf{c}) = \det(\mathbf{a}|k\,\mathbf{b}|\mathbf{c}) = \det(\mathbf{a}|\mathbf{b}|k\,\mathbf{c}) = k\,\det(\mathbf{a}|\mathbf{b}|\mathbf{c}) \tag{21}$$

gives (15).

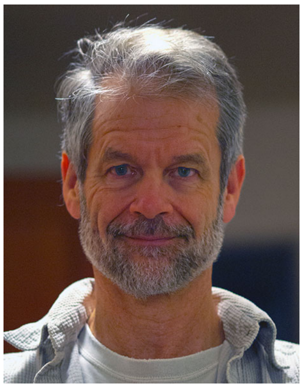
References

1. Adobe: Bake mesh maps. Substance 3D Documentation. https://helpx.adobe.com/substance-3d-painter/using/baking.html (2021)
2. Burley, B., Lacewell, D.: Ptex: per-face texture mapping for production rendering. Comput. Graph. Forum **27**(4), 1155–1164 (2008)
3. Cignoni, P., Montani, C., Rocchini, C., Scopigno, R., Tarini, M.: Preserving attribute values on simplified meshes by resampling detail textures. Vis. Comput. **15**(10), 519–539 (1999)
4. Cohen, J., Manocha, D., Olano, M.: Simplifying polygonal models using successive mappings. In: Proceedings of the 8th Conference on Visualization '97, VIS '97, pp 395–402 (1997)
5. Blender Foundation: Render baking. Blender 3.3 Reference Manual, https://docs.blender.org/manual/en/3.3/render/cycles/baking.html (2022)
6. Guidi, G., Angheleddu, D.: Displacement mapping as a metric tool for optimizing mesh models originated by 3D digitization. J. Comput. Cult. Herit. **9**(2) (2016)
7. Autodesk Help: Transfer maps. Autodesk Knowledge Network. https://knowledge.autodesk.com/support/maya/learn?s=Transfer+Maps (2021)
8. Jiang, Z., Schneider, T., Zorin, D., Panozzo, D.: Bijective projection in a shell. ACM Trans. Graph. **39**(6) (2020)
9. Lee, A., Moreton, H., Hoppe, H.: Displaced subdivision surfaces. In: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '00, pp 85-94, USA. ACM Press/Addison-Wesley Publishing Co. (2000)
10. Liu, L., Zhang, L., Yin, X., Gotsman, C., Gortler, S.J.: A local/global approach to mesh parameterization. Comput. Graph. Forum **27**(5), 1495–1504 (2008)
11. Maggiordomo, A., Ponchio, F., Cignoni, P., Tarini, M.: Real-world textured things: a repository of textured models generated with modern photo-reconstruction tools. Comput. Aided Geometric Des. **83**, 101943 (2020)
12. Marmoset: Baking in toolbag. https://marmoset.co/toolbag/baking/ (2022)
13. Panozzo, D., Puppo, E., Tarini, M., Pietroni, N., Cignoni, P.: Automatic construction of quad-based subdivision surfaces using fitmaps. IEEE Trans. Vis. Comput. Graph. **17**(10), 1510–1520 (2011)
14. Skala, V.: Barycentric coordinates computation in homogeneous coordinates. Comput. Graph. **32**(1), 120–127 (2008)
15. Tarini, M., Cignoni, P., Scopigno, R.: Visibility based methods and assessment for detail-recovery. In: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), VIS '03, pp 457–464 (2003)



Andrea Maggiordomo is a third-year PhD Student at the Computer Science department of the University of Milan, Italy, specializing in Computer Graphics and Geometry Processing. His main research interests focus on surface parametrization, texturing, and remeshing. He obtained his Master's Degree in Computer Science from the University of Pisa in 2018 and has previously worked as a Research Fellow at ISTI-CNR.

Yury Uralsky graduated from the Moscow State Technical University with degrees in computer science and electrical engineering before starting in the industry as a game developer. Yury has been with NVIDIA for over 18 years, where he held senior engineering and management positions and worked on several major GPU designs, including the latest Ada architecture. Yury is currently working on next-generation graphics hardware in the NVIDIA GPU architecture team. His interests include real-time rendering, computer architecture, programming models, and machine learning.



Henry Moreton (PhD 1992, Univ of California Berkeley) works as a Senior Distinguished Engineer at the NVIDIA Corporation. He has published in the areas of GPU architecture, curve and surface modeling, rendering, texture mapping, video and image compression, and unmanned submarine control. He has over one hundred patents in the areas of optics, video compression, graphics, system and CPU architecture, and curve & surface modeling & rendering. Current interests include mesh representations, the evolution of graphics programming models, API design, and the hardware architecture of highly parallel programmable devices.

Marco Tarini (PhD 2003, Univ of Pisa) works as a full professor at the Computer Science department of the University of Milan, Italy. A prolific researcher in the fields of Computer Graphics and of Geometry Processing, he has authored or co-authored more than 50 publications, the majority of which in top-tier journals of the field. His interests range in several aspects of geometry processing (surface parametrization and remeshing, 3D acquisition, shape reconstruction, tangent vector fields), of 3D real-time rendering (such as texturing and computer animation), and of their application in fields such as video-games, scientific visualization, virtual reality, and support for digital fabrication. He received several prestigious awards, including a "Young Researcher Award" by Eurographics in 2006, and a "Test of Times" award by Visualization in 2021.