Project no. **610886**

Project acronym: **ClinicIMPPACT**

Clinical Intervention Modelling, Planning and Proof for Ablation Cancer Treatment

Work programme objective: **ICT 2011.5.2 Virtual Physiological Human**

Instrument: **STREP**

Deliverable **D6.7**

Scientific publication on uncertainty visualization during RFA image processing and simulation in the clinical practice

Due date of deliverable: 31/07/2017

Actual submission date: 31/07/2017

Organisation name of lead contractor for this deliverable: **TUG**

Start date of project: **01/02/2014** Duration: **3 years**

Introduction

In this deliverable, we present a novel technique for computing texture coordinates of arbitrary surfaces on-the-fly in near-real-time frame rates. While this is not a visualization technique in itself, its sheer performance opens up many possibilities in the visual analysis field. The textures can be used, for example, to visualize positional uncertainty of the extracted surface, since these can never be computed exactly due to the usually low resolution of the underlying domain. It can also be used for multivariate visualization, for example, when mapping three distinct variables of the simulation simultaneously onto non-overlapping visual channels. The possibilities this technique enables is unfortunately too large to be exhaustively laid out in this article. Nevertheless, we provide a brief glance at what is possible.

On-the-fly Texturing of Implicit Surfaces

Philip Voglreiter, *Student Member, IEEE,* Michael Hofmann, Markus Steinberger, *Member, IEEE,* and Dieter Schmalstieg, *Senior Member, IEEE*

Abstract—We present a technique for generating parameterizations of implicit surfaces on the fly. Our method enables multivariate visualization directly on arbitrary surfaces by employing structural elements as additional visual variable. The flexibility of our method provides the opportunity to employ non-photorealistic techniques with respect to the geometry of a surface, direct application of small tileable elements, or geometry-driven glyph placement. Previous approaches to enhancing surface-based visualizations often only rely on local features, such as curvature, which makes global seamless distribution of texture elements difficult. Moreover, while many surface generation techniques produce incomplete meshes, e.g. only front faces, or entirely lack an explicit mesh representation, our approach makes do without extensive knowledge of the entire surface. In contrast to previous approaches for automatic texturing, our technique does not require time-consuming pre-processing steps or complex data structures. Thereby, our method is well suited for rapidly changing geometry, for example in time-varying data or in-situ visualization. Such scenarios further induce the need for smooth transitions between parameterizations of similar surfaces, since volatile visual changes may drastically increase the mental load of the viewer. We solve this issue via careful selection of reference points in object space, rather than on the surface directly. We additionally present explicit usage of the proposed techniques in texturing iso-surfaces resulting from direct volume rendering. Our approach is capable of generating full parameterizations of such surfaces almost in real-time and has a low memory footprint. This makes it ideal for usage in interactive applications.

Index Terms—Computer Society, IEEE, IEEEtran, journal, LaTeX, paper, template.

1 Introduction

Implicit surfaces often appear in visualization applications [1], [2], [3], [4]. Iso surfaces of volumetric simulations, high-opacity regions in medical volume rendering, or reconstructed surfaces from point clouds are widely used. However, typical visualization approaches do not use these surfaces to the full extent. While some techniques place glyphs on such implicit surfaces, most are restricted to, at most, using the color channel as additional visual variable.

In this article, we propose a novel technique for on-the-fly parameterization of such implicit surfaces. We exploit the resulting texture coordinates for providing an additional information channel by covering the entire visible surface in textural elements. Previous approaches [5], [6] to surface parameterization typically employ costly pre-processing steps. While this is a good choice for many scenarios, such as texturing 3D models, visualization applications must be able to cope with rapidly changing surfaces. For instance, simulation data often has a temporal component. Or, the user may want to browse through a wide range of iso-values. In such scenarios, the parameterization needs to be re-computed online. Moreover, for such rapidly changing surfaces, continuity in the parameterization is a key element. In both described scenarios, the surfaces usually only change slightly between consecutive frames. Hence, the parameterization should remain stable throughout these processes. In other words, the structural elements should not move erratically along the surface to avoid temporal visual artifacts and high mental strain for the user. Further, the parameterization should be continuous along the surface. Producing gaps in the visualization might lead to false conclusions from users and, consecutively, erroneous analysis results. And finally, placing (repeating) texture elements on the surfaces should be seamless, since artificial borders between single elements or surface patches might wrongly introduce visual structures that are not present in the data.

• *All authors were with Graz University of Technology, Austria, Graz. E-mail: last name @ icg.tugraz.at*

Manuscript received July 31, 2017

2 Related Work

We split the previous work into two parts. First, we deal with approaches to texturing surfaces automatically, and then we consider applications in visualization.

Lapped Textures [5] implement an idea that is similar to what we aim to achieve. The authors subdivide surfaces into patches and perform local parameterization. While the algorithm performs some local adaption in terms of orientation, the seams between the patches are apparent. The extension to 3D textures [7] goes into the direction of being able to re-parameterize the surface quickly, but requires user input, which we want to avoid.

Conformal maps [8], [9] also tackle the issue of texturing surfaces. Although these methods produce more accurate results and reduce the seams between surface subdivisions, their performance is far too low for interactive applications. While recent approaches [10] aim at optimized patch generation, all of these approaches suffer from strong seams between the single patches. The seams are usually not as apparent when using textures with an appropriately moderate frequency range at the lower end of the spectrum. We, however, aim at structural elements, which typically exhibit very high frequencies, i.e., strong edges.

More recently, focus shifted towards globally smooth parameterizations. Campen *et al.* [6] propose quantized global parameterizations. While the results are visually impressive, the computing duration of several seconds disqualify this

procedure for interactive scenarios. The extension of this approach [11] further optimizes the accuracy, but still does not provide the required performance for rapid surface changes.

In visualization, texture is often overlooked as visual variable, most likely due to the difficulties involved in actually applying them onto non-flat surfaces. In 2D, using texture elements for categorizing data, is rather straightforward [12]. For 3D surfaces, a few approaches exist. Splatting texture sprites [13] is comparably fast, but does not address the issue at the seams between sprites. Decal Maps [14] apply methods similar to our approach. The authors propose local parameterization via intersections with real, rasterized spheres in 3D. However, this approach does not tackle the global parameterization issue and instead only computes texture coordinates within these intersections. While this allows to place simple glyphs or locally delimited decals, using the same approach for continuous textures inevitable produces strong seams.

For generating a local parameterization, Schmidt *et al.* [15] proposed discrete exponential mapping. The basic idea is to compute a local geodesic map around a certain point on a surface. The geodesic distance can be computed incrementally, but lacks a sense of direction. The authors propose using user input for resolving this issue. Our approach strongly relies on the discrete exponential mapping approach, but modifies and extends it to achieve an interaction-free algorithm that, in contrast to the original, automatically generates the underlying directions and avoids seams by carefully placing the start points of the decals. In fact, the modifications lead to non-overlapping patches, computed in near-real-time.

3 Method

Let us briefly summarize the goals of the algorithm again before detailing the single steps. Most importantly, the parameterization needs to be computed on the fly. This indicates the need for massively parallel computation for all steps on top of general high performance of the employed algorithms. Despite parallel computing of texture coordinates over the surface, no seams should be created when these regions meet up. Secondly, the parameterization must cover the entire visible surface. Often, no explicit surface and neighborhood information exists, so we need to be able to generate at least an intermediate representation on the fly. The third requirement relates to interaction. The surface to be used as visualization medium may change rapidly, either during camera interaction or when the mapping from the data domain changes, e.g. when changing an iso-value or timestep in temporal data. During such interactions, the parameterization must be continuous to avoid artifacts. In a nutshell, the parameterization must not 'break up' when the surface grows, the texture origin should remain roughly at the same point, and the orientation of the texture space along the surface must not change. The upcoming subsections detail all steps required in satisfying these requirements. We separate the procedure into two major stages: Surface & Feature Extraction and Coordinate Generation.

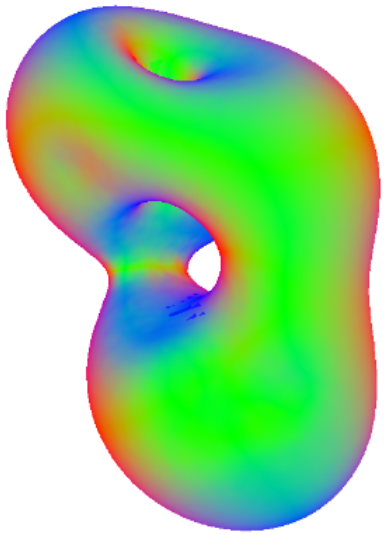

Fig. 1. A sample iso surface, extracted from a medical simulation. The color coding shows the surface normals. In practice, we have to anticipate difficult situations, such as in this case, where vessels penetrate the iso surface and leave a hole.

3.1 Surface & Feature Extraction

The first step of texturing the surface is, of course, extracting the surface itself. Throughout all upcoming procedures, we aim at the most general approach possible, so we assume that the surface itself is defined by a point cloud and no neighborhood information is present initially. For the remainder of this paper, we assume that this point cloud is generated by ray casting a volumetric dataset (Figure 1). This produces the densest, and consecutively most expensive to compute, representation. Other approaches, such as marching cubes [16] or more general point clouds like particle swarms [17] usually produce sparser representations. This effectively leads to better computational performance, but we will show that our approach is fast enough for the most complex scenario.

During ray casting, we already take care of the required consistent orientation of the parameterization and a subdivision of the surface for parallelization of the coordinate generation.

3.1.1 *Surface Ray Casting*

We use a standard, GPU-based ray casting approach for the volumetric data. Adaptive subsampling produces smooth, accurate results. Besides the surface hits themselves, we also store other primitives, such as gradient-based surface normals, for later stages.

3.1.2 *Direction Field*

For orienting the texture space on the surface independently from viewing parameters, we need to generate a notion of direction for the upcoming procedures. This has to fulfill two properties: Firstly, the direction field must be smooth, subject to the surface properties. This means that genus changes, as well as other discontinuities must be preserved. Further, similar to the texture coordinates themselves, we require the direction field to be insensitive to small changes of the surface.

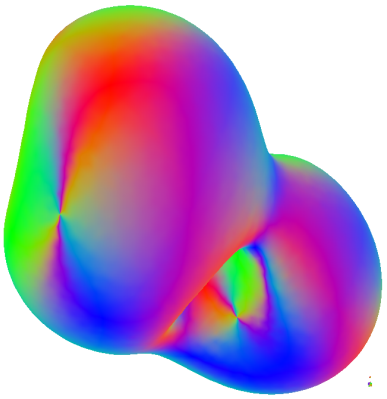

Fig. 2. The direction field we generate during ray casting. Note that points with a normal pointing in the direction (1,1,1) result in polar points. Nevertheless, even around these points, the field is smooth and allows for soft transitions.

We generate this direction field in relation to a global cartesian coordinate system described by three orthogonal basis vectors b_0, b_1, b_2. For each surface hit at pixel (x, y), we generate the corresponding normal $n_{x,y}$ via central differences. We then proceed to project the b_i into the tangent plane (eq. 1) and correct the sign (eq. 2) to produce unidirectional vectors and blend them together (eq 3).

$$b_{i_{proj}}(x,y) = b_i - n(x,y) * < n(x,y), b_i > \quad (1)$$

$$b'_i(x,y) = b_i * sgn(< b_i, n_{x,y} >) \quad (2)$$

$$dir(x,y) = \sum_{i=0}^{3} b_{i_{proj}}(x,y) * (1 - < n(x,y), b'_i(x,y) >) \quad (3)$$

While this direction field is smooth (Figure 2) and true to the surface features, it has one disadvantage. It generates poles. When using all three cartesian vectors, the poles collapse to singular points where the direction collapses and fans out in all directions equally. However, this only appears at surface points where the normal is equal to the cross-diagonal of the cartesian cube. If using only two basis vectors, the singularities become similar polar lines, relative to the intersection with the corresponding diagonal plane in the cartesian cube.

3.1.3 Patch Subdivision

The subdivision of the surface into patches is crucial for efficient parallel computing. Let us have a brief look at the requirements again: The patches should be of roughly equal size and evenly distributed over the entire visible surface. The second part of the requirements ties to changes of the surface. Under the assumption that the surface does not drastically change inbetween two rendered frames, the patches should remain similar in terms of size, position and distribution. If the surface grows or shrinks, e.g. for time-dependent data in in-situ visualizations, or due to changing the iso value, the subdivision should not change abruptly.

We cater to all these goals by imposing the patches with relation to the volumetric 3D space, rather than only with respect to the surface. To this end, we first need to define a 3D subdivision. The surface subdivision is then equal to the intersections with the single grid cells. Ideally, the chosen subdivision can be computed statically to minimize memory consumption and required computation. However, a simple regular grid subdivision is not ideal, since it does not evenly subdivide the surface as required.

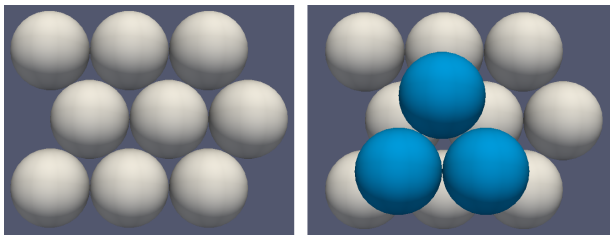

Fig. 3. Constructing a HCP grid. The virtual spheres in the rows touch each other and form a horizontal line. To form a layer, every other row is shifted to the right and in y direction so that it exactly fits into the gaps of the previous row. The second layer is then again shifted in x and y direction such that, after the shift in z, the 'spheres' exactly fall into the gaps of the previous layer again.

Fig. 4. Intersecting the surface with a HCP grid. The patches (random colorization) consist of two parts: The bright ellipsoid regions are direct intersections of the surface with a virtual sphere in the HCP grid. The dark regions around them still relate to the same node, but in the sense of proximity similar to a Voronoi diagram. We marked the borders between the patches in red and intersections where more than 2 neighbors meet with green pixels.

We instead chose the metaphor of hexagonal close packing (HCP) of equal spheres. In a nutshell, a HCP grid is a way to fill a certain volume with equally sized spheres optimally in the sense that the largest possible ratio of this volume is occupied. This can be achieved in layers: The first layer packs a sphere in x and y direction such that their centers form a lattice of equilateral triangles with side length equal to the radius of the spheres. The second layer is constructed equally, but shifted entirely such that the spheres 'fall' into the gaps of this first layer (Figure 3). The third layer is then again equal to the first, but offset in z direction.

Such a grid can be constructed as follows. Assuming a radius r that is constant throughout the grid, the cartesian

coordinates of a sphere center with grid index (i, j, k) are given by

$$x = (2 * i + ((j + k)\%2)) * r$$
$$y = (\sqrt{3} * j + \frac{1}{3} * (k\%2)) * r$$
$$z = (\frac{2 * \sqrt{6}}{3} * k) * r$$

We can now generate the subdivision of the surface by finding the closest sphere center for each surface hit. Due to the modulo operation and interdependent computation, inverting above equations has no closed-form solution. However, we can omit the modulo during inversion and perform a search in the immediate neighborhood of a candidate. The simplified inversion for any point $p(x, y, z)$ results in

$$k = \frac{3 * z}{(2 * r * \sqrt{6}}$$
$$j = \frac{y}{(\sqrt{3} * r)} - \frac{1}{3} * (round(k)\%2)$$
$$i = \frac{x}{2 * r} - \frac{1}{2} * (round(k) + round(j)\%2)$$

Rounding the grid index (i, j, k) then gives an approximation for the closest sphere center, but we nevertheless need to check the neighbors whether any of them is actually closer. Please be aware that this also generates patches which actually do not intersect the spheres. The spheres only serve as metaphor for subdividing the space evenly, while we are rather interested in the subdivision similar to a Voronoi diagram, where clear borders separate affiliation to one of the grid points. In Figure 4, bright regions denote the direct intersections with spheres and dark regions the additional Voronoi parts. Patches lacking the bright region do not directly intersect a sphere.

3.2 Parameterization

The previous step generated basic information of the surface that we now use to generate a parameterization. Let us summarize the requirements for the texturing approach that, at this point still need to be fulfilled. So far, we produced an even subdivision of the surface into patches and constructed a notion of direction during ray casting. We now need a way to enable parallel texture coordinate generation in all patches simultaneously. To achieve this, we require appropriate starting points in relation to the generated patches. Further, we simultaneously need to avoid seams in the texture coordinate that can occur between the patches. Distributing texture coordinates from multiple points towards each other can result in discrepancies when two or more wavefronts meet. Consecutively, each starting point for parallel coordinate generation requires appropriate start coordinates. Generating these start coordinates in itself is a diffusion process. It requires an absolute starting point for the entire texture and a way to compute start coordinates for any neighbor of an already computed one. Hence, we subdivide the description of this procedure into multiple steps.

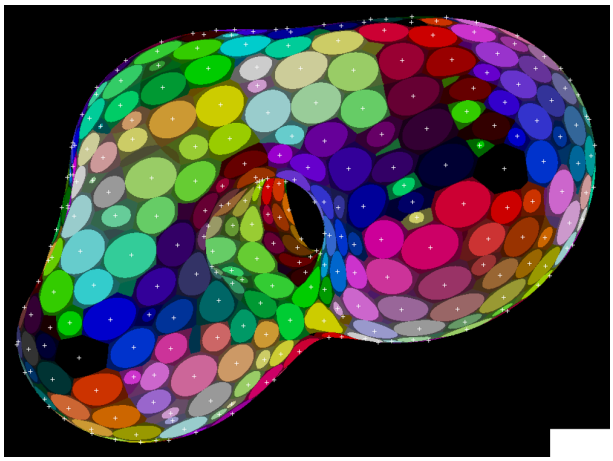

Fig. 5. For each patch that directly intersects a virtual sphere, we compute a reference point. We select the pixel within a patch that is closest to the virtual sphere center. For 'ghost nodes', i.e. those patches that project on the surface without a direct intersection of the corresponding sphere, we omit this computation.

3.2.1 Reference Points

We are basically looking for stable points on the surface to act as starting points for texture coordinate diffusion. In this stage, we can finally fulfill the requirement of stabilizing the coordinates during (minor) surface changes.

In Section 3.1.3, we introduced the surface patch generation scheme via HCP grid subdivision of the 3D space. An important observation for these patches is that the underlying grid stabilizes them quite well on the surface, even for larger changes. This, however, depends on the imposed resolution of the HCP grid. As long as the surface does not traverse an entire HCP grid node within two consecutive frames, the patch remains mostly stable. Over time, patches can of course appear or vanish, but this is typically not an abrupt change.

This allows us to generate a set of reference points on the surface. We compute these by finding the pixel within a patch that is closest to the corresponding sphere center, which can be efficiently computed in parallel. However, we need to be careful about another observation. Depending on the surface structure, patches may exist that do not directly intersect one of the spheres. In fact, patches consist of up to two internal regions: One denotes the intersection with the virtual sphere of the HCP grid, while this is surrounded by a region that does not intersect this sphere, but belongs to the patch in the Voronoi subdivision sense. However, due to the construction of the grid, the direct intersection region does not necessarily exist. We dubbed the HCP nodes these patches relate to as 'ghost nodes', as they either intersected a sphere in one of the previous frames, or might in one of the upcoming iterations. We observed that the minimum distance computation for these ghost nodes is difficult for various reasons. The first one is that surfaces in these regions often exhibit strong features, e.g. large curvature. The second observation is that, after intersection with the surface, these nodes may project onto two or more disconnected regions on the surface. While such nodes could theoretically form several distinct surface patches for the upcoming parameterization, treating them

this way is computationally much more expensive. Hence, we do not allow ghost nodes to spawn reference points as starting points for the texturing, but keep them as interfaces for the upcoming start coordinate distribution.

3.2.2 Reference Point Triangulation

Now, before being able to compute the start coordinates for the reference points (Section 3.2.1), we require general rules for distributing the coordinates over the entire surface. We therefor perform a parallel triangulation of the reference points as extracted from the surface patches.

This initially requires generating neighborhood information. This can be computed easily in screen space by checking each direct neighbor of each pixel in parallel. If a neighboring pixel belongs to a different patch, these two patches are considered as direct neighbors. We compute all direct neighbors for all reference points to form potential triangle edges. The third reference point that completes the triangle then needs to be a mutual neighbor of both previous points.

Since we compute all possible triangles of all reference points in parallel, we need to additionally impose filters to result in non-overlapping and non-intersecting triangulations in a deterministic fashion. After forming the first edge, several possibilities arise with the remaining mutual neighbors. We filter these out for exactly one triangle on the 'left' and 'right' side of the edge in image space. This filter only considers the triangle, per side, that has the smallest circumference. This results in emitting the same triangle exactly three times, once per vertex, but discarding all other potentially overlapping or intersecting primitives. The final filter then computes the linear grid index of the evoking reference point. If and only if this is the reference point with the smallest grid index, it emits the triangle.

However, the ghost nodes play an important role again. If we would treat them as standard HCP grid nodes and use their corresponding patches, the possibility of dual or multiple distinct surface projections per node would unnecessarily complicate the procedure of sorting out the 'correct' triangles. In fact, comparably distant HCP nodes might be treated as neighbors and form an multiple intersecting triangles. However, we can use the ghost nodes as interfaces to connect reference points to a neighbor beyond the ghost node's projections. In practice, ghost nodes are typically not direct neighbors of each other. Although probably possible, we did not encounter a single case where a ghost node was not entirely surrounded by standard reference point patches. For triangulating under these circumstances, we need to make a few additional observations.

A ghost node can be surrounded by three to six reference point patches (Figure 6). We now need to filter the possible triangulations of the enclosing polygon to result in a deterministic outcome (remember, all reference points compute their respective triangles simultaneously). This boils down to identifying identical edges that cross the ghost node. In case of four neighbors, one of two possible crossing edges needs to be selected, while for five neighbors, two out of five edges are relevant, and for six, three out of seven, respectively.

First, we need to identify the current neighbor of a standard node a ghost node. We then reconstruct all crossing

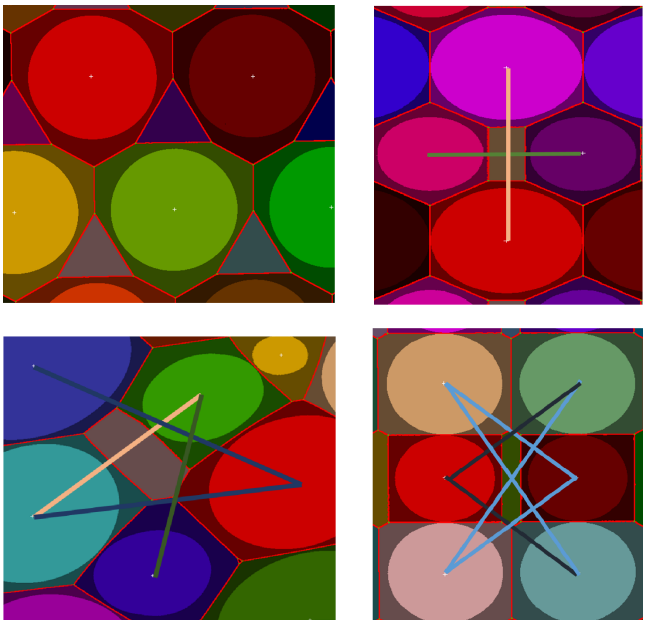

Fig. 6. Possible configuration for neighborhood of ghost nodes. While three neighbors (top,left) is a trivial case, all other configurations require careful determination which crossing edges should be used for parallel triangulation purposes. Using the shortest cross edge as starting point, we are able to resolve this issue deterministically for any reference point involved.

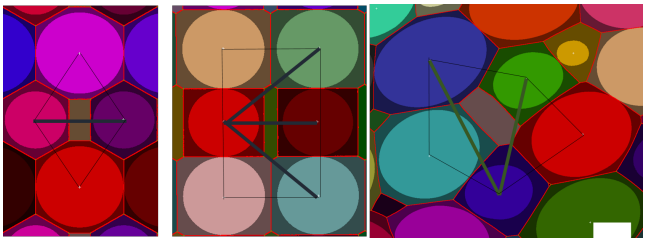

Fig. 7. After finding the shortest crossing edge, we can determine the remaining edges relevant for a full triangulation of this neighborhood. We determine the node of the shortest cross edge with the smaller linear HCP grid index to be responsible for all remaining cross edges as well, leading to a deterministic resolution of this issue.

edges after sorting the neighbors of this ghost node. Then, we pick the shortest crossing edge and determine the vertex whose reference point corresponds to the HCP node with the smaller index. We then exploit the observation that a triangulation of a polygon surrounding a ghost node can be generated by drawing crossing edges from a single vertex. Each node, during parallel computation, then tries to find a configuration which results in triangles involving one of those cross edges, and proceeds as described before (Figure 6).

This ultimately leads to a comparably fast parallel triangulation (Figure 8) procedure that avoids intersections and overlaps. The edges of the triangulation then serve as information channels for the initial start coordinate distribution.

3.2.3 Start Coordinates

Exactly computing the start coordinates for all reference points is only possible when already knowing the entire parameterization of the surface. Since this is contradictory

Fig. 8. Triangulation of the reference points. The imposed heuristics produce a complete, non-overlapping triangle mesh between the reference points in parallel, which leads to extremely fast computation.

to the idea of our approach, we propose an approximation. Doing so requires a few assumptions, though. When treating this initial coordinate distribution as a problem of geodesic coordinates, we must assume that the edge between two reference points is simultaneously the shortest one. Using a high enough resolution of the underlying HCP grid inherently solves this problem. We must also assume that, between two reference point, the genus of the surrounding surface does not change. In other words, no holes or manifolds must be present. The former issue can be resolved by postponing it to the coordinate filling step and resuming the coarse distribution nonetheless. Since we are implicitly assuming iso surface properties, the latter requirement automatically holds for the triangulation procedure we previously described.

In the first step, we need to determine the absolute starting point for the coordinates. Experiments showed that the most suitable candidate is the reference point whose corresponding HCP grid node has the most surface hit points. This implicitly excludes nodes with an intersection area that does not approximately face the camera, since these are generally hit by only few rays. Further, these nodes usually project rather centrally on the screen due to the perspective projection. We then distribute the start coordinates in a diffusion process from the initial point over the entire triangulation along the edges.

The edges themselves do not faithfully reproduce the surface behavior. For the accurate arc length, we reproduce the actual surface positions via reprojection of the points along the edge during incremental sampling. The direction field (Section 3.1.2) further aids this process. Since this vector field is smooth, we can incrementally sample along the edge and compute piecewise linear distances and, accordingly, the appropriate vector.

We again exploit parallelism in this procedure. In each iteration, each reference point that has no start coordinates yet checks whether any neighbors already do. It then computes the distance as described above for all available neighbors and chooses the shortest one. The performance of this procedure is of course closely tied to the underlying grid resolution, but in practice shows high performance.

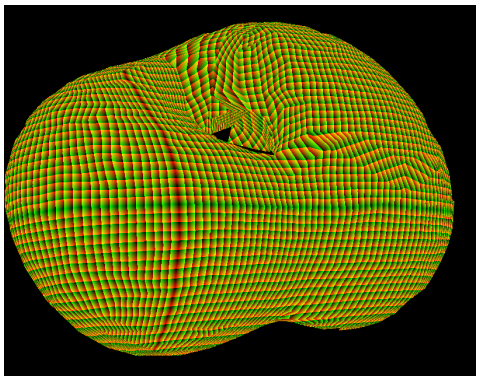

Fig. 9. Final computation of the texture coordinates after executing the entire workflow. Note that there are still minor errors, depending on the complexity of the surface. However, the excellent performance of the algorithm appears to be a good trade-off.

3.2.4 *Texture Coordinates*

Now that each reference point has appropriate start coordinates, we can parallelize the parameterization process. We use a slightly modified version of discrete exponential mapping [15]. In its original idea, this algorithm computes geodesic distances over a surface in conjunction with Dijkstra's shortest path algorithm [18]. However, in our scenario, this is much too expensive; We treat every pixel on screen as distinct node to improve the surface reconstruction accuracy and keep fine details. This would lead to a connected graph for all hit pixels on which we would have to run Dijkstra's algorithm. Instead, we implement a greedy variant. Since our patches are comparably small, we can safely assume that no strong surface features exist within each patch.

The procedure consists of two steps. First, we initialize the immediate neighborhood of each reference point in parallel. Then, we again iteratively diffuse these coordinates outwards. This includes several considerations: First, we treat this as the problem of filling the triangles spanned by the reference points. Since the edges serve as stopping criteria, this approach balances the computation over the entire surface. Additionally, this minimizes error accumulation during the diffusion process since each start coordinate only contributes to a small ratio of the entire surface. We allow each triangle to diffuse its start coordinates from all vertices. By using the median of the three computed components, we further reduce the error introduced by the coarse computation of start coordinates (Figure 9).

Similar to the initial distribution of approximate start coordinates, we employ a greedy strategy. In each iteration, every pixel with a surface hit checks whether any of the direct neighbors already has computed its coordinates. It then performs a single step of discrete exponential mapping for each neighbor and selects the result that exhibits the shortest distance in texture coordinates. The number of iterations required to fill the entire surface strongly depends on two factors. For one, the HCP grid resolution dictates the size of the projected patches and thereby also of the triangles.

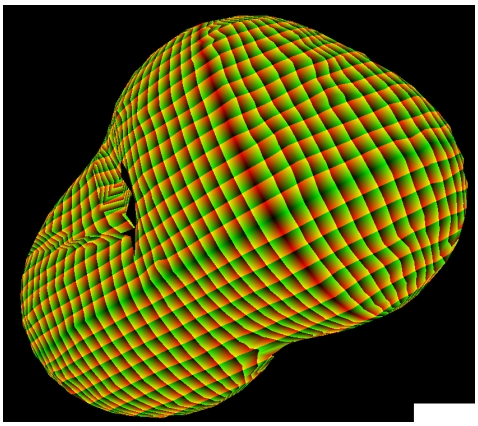

Fig. 10. We automated almost the entire approach. However, one parameter still needs to be determined by the user: The scale factor. This is impossible to automatically compute, since we cannot make any assumptions on the textures the user will later on map onto the texture.

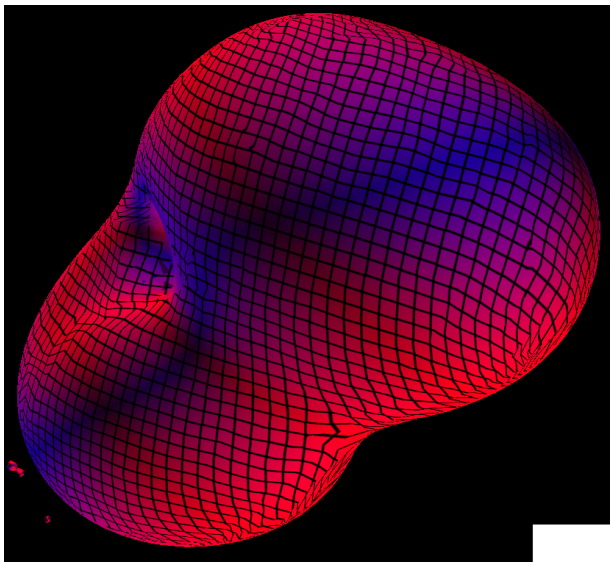

Fig. 11. A simple application of the proposed parameterization technique. We extract an iso surface from a volumetric dataset. We then map the positional uncertainty of this iso surface in the red-to-blue color spectrum, and apply a simple structure texture. While real visualization applications will exploit the technique more thoroughly, this simple figure shows the most critical aspect: We can generate an almost seamless parameterization of implicit surfaces on-the-fly.

Second, a good balance between the size of the initialization and filling procedure influences the performance. However, this also further depends on other factors, such as the camera parameterization and how large the projections of the patches are on the screen. If only a few reference points are visible under a certain configuration, computing more pixels in the initialization procedure becomes more costly. However, we will discuss more details in the implementation section.

However, a small error remains: For surfaces with a higher genus, e.g. a hole (Figure 10), the coordinates distribute in both directions around this hole due to the greedy procedure. This will lead to a minor discrepancy where the two computation strains meet up.

3.3 Visualization with Textures

Now that we have a smooth parameterization of the surface, we can use it for actual visualization purposes. Standard techniques would typically encode two visual variables for surface visualization purposes: The first scalar value in a multivariate scenario maps onto the position of, e.g., an iso-surface. A second variable can then occupy the color channel. The parameterization, however, now enables multivariate visualization. There are multiple possibilities. As shown in Figure11, we can map structural elements onto the surface. By categorizing a scalar field and using easily distinguishable patterns [12], multivariate analysis of simulation data becomes possible. Additionally, when also incorporating opacity, this approach could encode factors such as uncertainty.

However, there are many other possibilities we do not explore in depth in this article: The subdivision and reference point generation can serve as anchor point for glyph visualizations [19], [20] under even distribution. Further, combining the starting points and the direction field can be used for hatching and stippling [21] techniques, where the parameterization serves as scaling property.

4 Implementation

Implementing all steps of this method requires careful optimization to achieve interactive frame rates in larger resolutions. With the steps described below, texturing surfaces with 30 frames per second is possible with modern hardware. The upcoming sections divide the implementation into the logical render passes, which slightly differs from the method description in terms of ordering.

Since we parallelize all necessary tasks, almost no memory transfer between GPU and CPU is necessary, except for single counters.

4.1 Surface Ray Casting

We implement a standard iso-surface ray casting approach. First, we rasterize the bounding box and store front and back faces to determine entry and exit points for the rays. The rays sample through the volume and detect whether the volume intensity crosses a set iso-value in a step. If this is the case, we perform binary sub-sampling in this range to more accurately approximate the surface position. From these hit points, we also compute the normal using central differences.

In the same render pass, we also compute the direction field and the surface patches. The former is straight forward applying the equations given in Section 3.1.2. The surface patches are a bit more complex, since we want to simultaneously determine which patch a pixel belongs to and also whether it represents the reference point for this patch. We first compute which patch the pixel belongs to (Section 3.1.3) with the given equations. Determining the reference point then requires finding exactly that pixel of a surface patch that has the minimal distance to the corresponding HCP grid point. We store this information in an SSBO, where each entry corresponds to one HCP grid node. We need to store both the distance and the corresponding pixel for each of them. However, since locking mechanisms rarely work reliably in OpenGL implementations, we implement this

minimum finding problem differently. OpenGL supports 64 bit arithmetic for integers, including atomic operations. We concatenate the distance measure after transforming it into a 32 bit integer with the screen coordinates (2x16 bit) into a 64 bit integer. Since the most significant 32 bit are reserved for the distance, we can use atomic minimum functions to determine the reference points for each patch during ray casting in parallel.

In addition, we employ atomic counters, one per HCP node, to store the number of pixels referring to each. This is relevant for later determination of the absolute start point of the texture coordinates.

4.2 Surface Patches

This section contains all render passes related to triangulating the reference points and establishing the start coordinates.

4.2.1 Neighborhood

First, we need to establish all neighborhood relations with a single call of a compute shader. For each pixel that contains a surface and relates to a HCP grid node, we first determine which node it belongs to. Then, we check all neighbors of this pixel. If they correspond to a different node, we store the mutual neighborhood relation in an SSBO that can hold up to 12 neighbor entries per HCP grid node. We also avoid duplicate entries. Although these would not lead to wrong results, the triangulation performance would suffer.

We experimented with triangulating directly in object space with relation to the HCP grid. However, resolving all possible configurations was much more expensive than the image-based variant.

4.2.2 Relative Edge Coordinates

Using the neighborhood information, we can now employ a compute shader that computes the relative texture coordinate information between reference points. Each active point employs the incremental exponential mapping procedure, but limited to the direct path between the two points. While the straight edge would neglect important surface data, we reproject a number of sample points along the planar edge back onto the actual surface. This, in practice, produces much more accurate results.

We compute this for all connected pairs, regardless of the upcoming triangulation since its influence on the parameterization is limited to subdivision tasks. Producing the complete information also allows us to more accurately exploit the greedy distribution procedure described previously.

Moreover, we determine which reference point is the best candidate for absolute zero in terms of texture coordinates. Since we previously stored the pixel count for all patches, we now only need to perform an atomic maximum operation for all reference points in parallel.

4.2.3 Start Coordinates

Using the absolute starting point and relative coordinates from the previous step, we now compute the coarse initial coordinate distribution, providing local starting points for each reference point. This boils down to applying exponential mapping with 'long' steps, i.e., the whole edge.

This is an iterative procedure in a compute shader. In each iteration, every active HCP node and its respective reference point checks whether its neighbors have already been computed. If at least two are available, we select the one with the relative edge coordinates with the shorter distance. Then, we perform the exponential mapping step and concatenate the edge information with the previous starting point.

The number of iterations this procedure requires depends on the resolution of the HCP grid and the location of the absolute starting point. However, since we choose this point as the patch with the most contained pixels, it is usually rather central on the surface projection. Hence, the number of iterations is rather small.

4.2.4 Triangulation

The triangulation uses only a single compute pass. For each active reference point, we compute the possible triangle it can create as described in Section 3.2.2. We store the triangles in an SSBO we can then re-bind in OpenGL for rendering usage. We then rasterize the triangles and create a texture with triangle IDs that we can then use for delimiting the diffusion process of texture coordinates.

4.3 Parameterization

The parameterization consists of two compute passes. In the first stage, we initialize the immediate neighborhood of all reference points. In total, we fill each triangle three times, starting from each vertex. This requires using three textures, one per component. We select the appropriate component via barycentric in the initialization step: If any of the three barycentric interpolation parameters is close to 1, we can decide which of the three textures to write to (and read from). Besides two floats for the texture coordinates, we also store the reference point coordinates as integers for quick lookup, totaling to 4 values per pixel, ideally fitting standard OpenGL layouts.

For initialization, we launch one thread per active HCP grid node. The 1-ring around the corresponding reference point pixels cannot be computed using discrete exponential mapping, so we simply project the object space positions in to the tangent plane of the reference point. We use the direction field for orientation and incorporate the start coordinates from the previous stages as local origin. For stable computation, we initialize a square of 11x11 pixels centered at the reference point. Otherwise, it could be possible that the upcoming filling stage fails for triangles with acute angles.

The filling stage performs region growing according to the initialized coordinates from the first stage. This requires launching a compute shader with one thread per pixel until the entire surfae is filled, or if the previous iteration filled only a few pixels, indicating that the algorithm is almost complete.

Each pixel, per iteration, performs the following steps. First, it looks up its triangle ID. It then tries to compute each of the three overlapping texture coordinates. We require each pixel to find at least three computed neighbors in the respective triangle. Since we perform greedy exponential

mapping, this reduces the error introduced by the assumptions drastically, since the local smoothness criterion guarantees that one of these neighbor pixels is a good enough approximation. Requiring more than three neighbors to be present is not possible in practice. Either, this drastically reduces the performance, or the algorithm does not converge in large areas due to the triangle heuristic.

Once we computed all three parameterizations for all triangles, we finally average the coordinates with additional outlier detection. In practice, growing the parameterization from all three directions leads to much better results compared to computing only a single one.

4.4 Performance

Due to the wide ranging parallelization effort of all subtasks, we achieve high computational performance. We tested the algorithm on a GForce GTX 1070 GPU. For a resolution of 1200x800 pixels, and a HCP grid resolution of 32^3, the entire workflow takes 25-50 milliseconds (ms).

The first step, involving ray casting, direction field generation and surface patch creation, takes about 10-13 ms and thereby amounts to a comparably large portion of the overall duration. The most costly part here is the ray casting itself, whereas the two additional steps are only 5-10% overhead.

The second stage, in which we search for neighbor relations between the surface patches and perform the triangulation, is negligible at 1.0-1.5 ms.

Computing the relative coordinates for the edges and distributing these as start coordinates from the absolute origin is a bit more expensive again, but at typically less than 10 ms still very fast.

Triangle rasterization takes significantly less than 1 ms and is therefore negligible. Initializing the coordinates of a 11x11 neighborhood at the reference points hovers around the same performance. Unsurprisingly, the coordinate diffusion procedure is a bit more expensive again. Depending on the triangle distribution and shape, this can require up to 80 iterations for the described configuration. Nevertheless, this final step only takes about 15 ms.

Overall, the computational performance of the algorithm, is extremely high. We can create parameterizations for implicit surfaces at the border between interactive and real-time frame rates. Please note that these measurements do not take potential improvements, such as warping coordinates from frame to frame rather than re-computing them, into account.

5 Discussion

We presented a collection of algorithms that enables parameterization of implicit surfaces almost in real-time. By exploiting parallelism throughout the entire workflow, the presented technique optimally exploits modern GPU capabilities. We believe that this is an important toolset for visualization tasks, for instance for visualizing additional variables on iso surfaces, optimal glyph placement, uncertainty visualization or non-photorealistic techniques such as hatching or stippling.

However, the approach currently does not support detection of more complex surface features. In some cases, iso surfaces can fold over multiple times along the view direction. We consider this as an interesting future direction of research, since it involves more complex surface analysis. Another possible starting point for additional investigation is how transparency and multiple blended iso surfaces would behave. Simple blending is known to cause situations that are difficult to assess since the human cognition cannot separate colors easily. However, the structure textures may aid the separability of consecutive surfaces. Nevertheless, this could also lead to visual clutter.

Acknowledgments

This work was co-funded by the European Union in the projects ClinicIMPPACT (grant no. 610886) and GoSmart (grant no. 600641).

References

[1] C. B. Hübschle and P. Luger, "Moliso–a program for colour-mapped iso-surfaces," *Journal of applied crystallography*, vol. 39, no. 6, pp. 901–904, 2006.

[2] P. J. Rhodes, R. S. Laramee, R. D. Bergeron, T. M. Sparr *et al.*, "Uncertainty visualization methods in isosurface rendering," in *Eurographics*, vol. 2003, 2003, pp. 83–88.

[3] M. Herrmann, "Visualizing turbulent flames using flamelet libraries," *Combustion and Flame*, vol. 175, pp. 237–242, 2017.

[4] L. Liu, D. Silver, K. Bemis, D. Kang, and E. Curchitser, "Illustrative visualization of mesoscale ocean eddies," in *Computer Graphics Forum*, vol. 36, no. 3. Wiley Online Library, 2017, pp. 447–458.

[5] E. Praun, A. Finkelstein, and H. Hoppe, "Lapped textures," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 465–470. [Online]. Available: http://dx.doi.org/10.1145/344779.344987

[6] M. Campen, D. Bommes, and L. Kobbelt, "Quantized global parametrization." *ACM Trans. Graph.*, vol. 34, no. 6, pp. 192–1, 2015.

[7] K. Takayama, M. Okabe, T. Ijiri, and T. Igarashi, "Lapped solid textures: Filling a model with anisotropic textures," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 53:1–53:9, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1360612.1360652

[8] B. Lévy, S. Petitjean, N. Ray, and J. Maillot, "Least squares conformal maps for automatic texture atlas generation," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 362–371, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/566654.566590

[9] Y.-W. Miao, J.-Q. Feng, C.-X. Xiao, Q.-S. Peng, and A. R. Forrest, "Differentials-based segmentation and parameterization for point-sampled surfaces," *Journal of Computer Science and Technology*, vol. 22, no. 5, pp. 749–760, Sep 2007. [Online]. Available: https://doi.org/10.1007/s11390-007-9088-5

[10] Y. Cao, D.-M. Yan, and P. Wonka, "Patch layout generation by detecting feature networks," *Computers & Graphics*, vol. 46, pp. 275 – 282, 2015, shape Modeling International 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849314001101

[11] M. Campen, M. Ibing, H.-C. Ebke, D. Zorin, and L. Kobbelt, "Scale-invariant directional alignment of surface parametrizations," in *Computer Graphics Forum*, vol. 35, no. 5. Wiley Online Library, 2016, pp. 1–10.

[12] P. Voglreiter, M. Hofmann, C. Ebner, R. Blanco Sequeiros, H. R. Portugaller, J. Fütterer, M. Moche, M. Steinberger, and D. Schmalstieg, "Visualization-guided evaluation of simulated minimally invasive cancer treatment," in *EG Workshop on Visual Computing in Biology and Medicine*, 2016.

[13] S. Lefebvre, S. Hornus, and F. Neyret, "Texture sprites: Texture elements splatted on surfaces," ACM SIGGRAPH, April 2005. [Online]. Available: http://www-evasion.imag.fr/Publications/2005/LHN05

[14] A. Rocha, U. Alim, J. D. Silva, and M. C. Sousa, "Decal-maps: Real-time layering of decals on surfaces for multivariate visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 821–830, Jan 2017.

[15] R. Schmidt, C. Grimm, and B. Wyvill, "Interactive decal compositing with discrete exponential maps," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 605–613, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1141911.1141930
[16] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *ACM siggraph computer graphics*, vol. 21, no. 4. ACM, 1987, pp. 163–169.
[17] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel, "Feature sensitive surface extraction from volume data," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001, pp. 57–66.
[18] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec 1959. [Online]. Available: https://doi.org/10.1007/BF01386390
[19] M. O. Ward, "A taxonomy of glyph placement strategies for multidimensional data visualization," *Information Visualization*, vol. 1, no. 3-4, pp. 194–210, 2002.
[20] T. Ropinski, S. Oeltze, and B. Preim, "Survey of glyph-based visualization techniques for spatial multivariate medical data," *Computers & Graphics*, vol. 35, no. 2, pp. 392–401, 2011.
[21] M. Hummel, C. Garth, B. Hamann, H. Hagen, and K. I. Joy, "Iris: Illustrative rendering for integral surfaces," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1319–1328, Nov 2010.