Sample-based Surface Coloring

Kai Bürger, Jens Krüger, Rüdiger Westermann

Abstract—In this paper we present a sample-based approach for surface coloring, which is independent of the original surface resolution and representation. To achieve this, we introduce the Orthogonal Fragment Buffer (OFB)—an extension of the Layered Depth Cube—as a high-resolution view-independent surface representation. The OFB is a data structure that stores surface samples at a nearly uniform distribution over the surface, and it is specifically designed to support efficient random read/write access to these samples. The data access operations have a complexity that is logarithmic in the depth complexity of the surface. Thus, compared to data access operations in tree data structures like octrees, data-dependent memory access patterns are greatly reduced. Due to the particular sampling strategy that is employed to generate an OFB, it also maintains sample coherence and thus exhibits very good spatial access locality. Therefore, OFB-based surface coloring performs significantly faster than sample-based approaches using tree structures. In addition, since in an OFB the surface samples are internally stored in uniform 2D grids, OFB-based surface coloring can efficiently be realized on the GPU to enable interactive coloring of high-resolution surfaces. On the OFB we introduce novel algorithms for color painting using volumetric and surface-aligned brushes, and we present new approaches for particle-based color advection along surfaces in real-time. Due to the intermediate surface representation we choose, our method can be used to color polygonal surfaces as well as any other type of surface that can be sampled.

Index Terms—Sample-based graphics, graphics data structures, surface coloring, surface particles

1 Introduction

In computer graphics, 3D surface coloring is an important technique for adding realism to computer-generated imagery and for generating a specific artistic context. Especially the possibility to interactively perform paint art on computer generated models, instead of merely scanning physical paintings and bringing them onto such models, has always been very desirable.

The transfer of color to a 3D polygonal surface can be performed by vertex coloring, i.e. by adding color to the polygon vertices and shading the polygons during rendering. Since the resolution of the surface in general does not reflect the color subtleties to be added, vertex coloring restricts the color variation to the surface resolution. To overcome this limitation the surface has to be refined up to the maximum resolution of added color details.

The transfer of color to a surface can also be realized in the domain of a surface parametrization, by adding color to the points in the 2D parameter space, i.e. the texture domain. This method, however, requires a satisfactory parametrization, i.e. a parametrization that has low distortion and thus yields a more uniform resolution of added color details on the 3D surface. For arbitrary surfaces it becomes very difficult to find such a parametrization, especially if adjacent 3D surface primitives should be mapped to adjacent regions in the texture map.

- *K. Bürger (E-mail: buerger@tum.de) and R. Westermann (E-mail: westermann@tum.de) are with the Computer Graphics & Visualization group, Technische Universität München*
- *J. Krüger (E-mail: jens.krueger@dfki.de) is with the DFKI Saarbrücken and the Scientific Computing and Imaging Institute, University of Utah*

A third strategy is to resample the surface into a spatial grid and to add color to affected grid cells. While a uniform grid allows for the efficient read/write access to these cells, it is too memory intensive in general to be of practical relevance. This problem can be cured by minimal perfect spatial hashing, which supports access to object samples via an injective hash function in expected time O(1). Unfortunately, computing such a function is not always possible, and even computing nearly minimal hash functions involves expensive preprocessing [1].

An adaptive grid, on the other hand, can greatly reduce the memory requirement, but it provides a less efficient interface to the data samples. Specifically, if the grid is encoded into a tree data structure like an octree, accessing the data that is stored in this structure generally exhibits the $O(log_2(N))$ complexity (with N being the grid size corresponding to the maximum refinement level). This access requires a sequence of pointer indirections, and each indirection depends on the result of the previous one. On current computing architectures this leads to memory dependent execution stalls and noticeable performance cuts.

2 Contribution

The primary focus of this paper is the development of an efficient method for surface coloring, which is independent of the surface resolution and does not need a surface parametrization. To achieve this, we use a spatial data structure that stores a resampled version of the surface—the Orthogonal Fragment Buffer (OFB). The OFB is conceptually similar to the Layered Depth Cube (LDC) introduced by Lischinski and Rappoport [2], which itself builds on Layered Depth Images (LDI) [3]. While a LDI captures all depth layers of an object

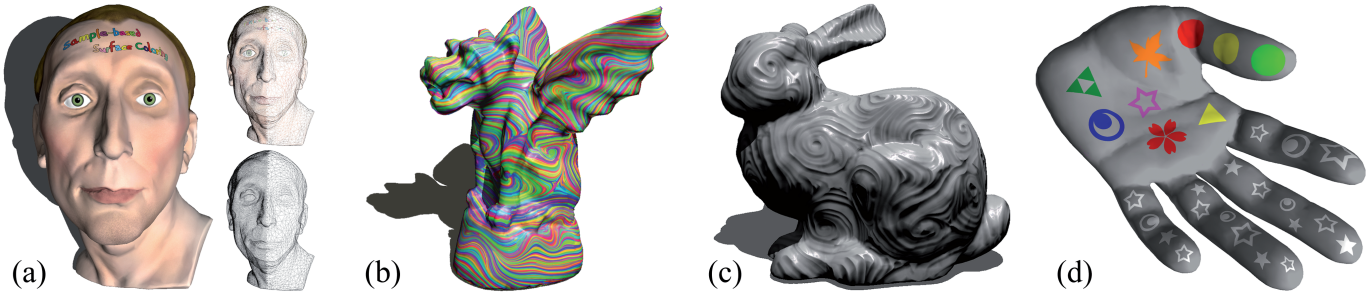

Fig. 1. Resolution-independent sample-based appearance modeling of polygonal surfaces: (a) fine color details are painted on an adaptive triangle surface, (b) particle-based color advection, (c) normal modulation, (d) decal painting using a surface-aligned brush. All effects were generated at update rates of 100 fps and higher on recent GPUs.

in the order they are seen from one particular direction, a LDC captures these layers from three mutually orthogonal directions, thus representing a surface point up to three times in the data structure. In our approach, the sampling is also performed along sets of parallel rays emanating from mutually orthogonal uniform 2D grids. Along these rays, however, only surface points with an angle less or equal to 45 degrees between the surface normal at this point and the ray direction are considered. In this way, redundant sampling of the same point is avoided, and a quasi-uniform sampling with a maximum distance foreshortening of $1/\sqrt{3}$ is generated. Thus, our data structure can be seen as a redundance-free LDC.

Since the sampling can be performed by coordinate projections into uniform 2D grids, the OFB can be seen as a hashing of surface points using the projections as hash functions. Due to the underlying regular grid structure, this hashing maintains sample coherence so that the OFB exhibits very good spatial access locality. However, since the hashing maps multiple samples onto the same grid cell, it is not perfect. Specifically, it produces up to d collisions per sample, with d being the surface's depth complexity. Since the samples falling into the same cell can be sorted with respect to their distance to the sampling grid, the computational complexity of finding an entry in the hash table is $O(log_2(d))$. On the other hand, if the samples falling into the same cell are not sorted, data-dependent memory access patterns—and memory latencies thereof—can be avoided entirely. Therefore, depending on the efficiency of data dependent memory access operations on the underlying hardware architecture, either a sorted or an unsorted OFB can be chosen flexibly.

Due to its properties the OFB is also well suited for the GPU, especially in scenarios where the massive parallelism available on the GPU can be exploited, i.e., where many data access operations have to be performed in parallel. Throughout this paper we will introduce a number of coloring effects where parallel read/write access is required, some of which are shown in Figure 1. We present algorithms for color and normal painting using spatially extended brushes, and we introduce particle-based approaches for simulating color advection along surfaces and surface-aligned paint brushes. This includes a novel and highly efficient technique to trace massive amounts of particles on an OFB. Figure 2 demonstrates the use of this technique to uncover a transparent surface via color transport along streamlines in a vector field given on this surface.

Due to the use of an OFB, our surface coloring method reduces performance limitations and vastly exceeds existing approaches with respect to speed and resolution. Even though at first glance the method seems to be exhaustive in terms of memory consumption, we will show that an adaptive tree data structure at comparable resolution requires only slightly less memory. This is in particular due to additional information that has to be stored in a tree structure to enable the efficient access to adjacent surface samples.

We should point out that our method is subject to typical limitations of resampling-based approaches, such as the loss of detail caused by an under-sampling of the surface or the blurring of sharp features like edges due to the regular sampling pattern. It is also clear that filtering in the OFB becomes significantly more expensive than in the 2D texture domain, since adjacent samples may reside in different sampling grids and at different positions in the sequence of samples that are captured in the respective grid cells.

The remainder of this paper is organized as follows: In Section 3 we discuss previous work that is related to ours. Next, we introduce the OFB and describe its internal structure. Section 5 describes the color transfer to an OFB. In Section 6 we introduce *surface particles*, which use the OFB to move on a surface along a given direction field. Section 7 demonstrates the use of surface particles to simulate a surface-aligned 2D brush. In Section 8 we present a highly efficient algorithm for OFB construction using rasterization hardware. Section 9 presents results, and Section 10 concludes the paper with a discussion of limitations of our work and future extensions.

3 Related Work

The computer graphics technique most closely related to ours is interactive surface painting. Surface painting techniques have been at the core of computer graphics

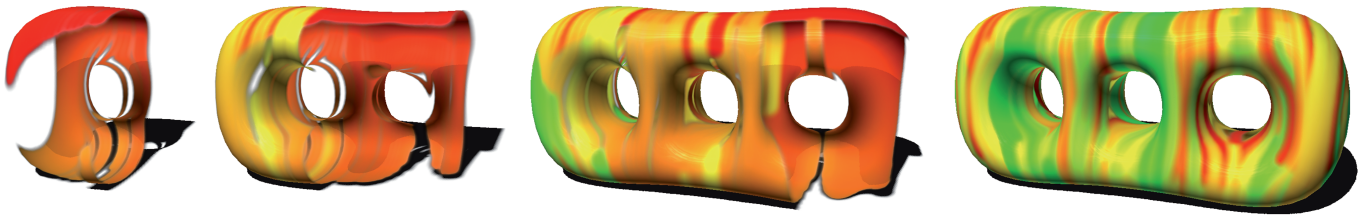

Fig. 2. A transparent surface is more and more revealed by particles moving on the surface and leaving color on it.

research for many years, and today there exists an extensive amount of literature on this subject which can be categorized into three different classes: *User interface*, *brush model*, and *paint transfer*.

Into the first category falls research on the user interfaces used to directly paint on 2D canvas or 3D shapes, ranging from simple 2D mouse-based devices to intuitive 3D haptic devices. In the second category, research is pursued on the computer simulation of realistic paintbrush models and their interaction with the surface. There is a vast body of literature related to these two categories and a comprehensive review is beyond the scope of this paper. However, Baxter et al. [4] and Adams et al. [5] discuss some of the devices and brush models used and provide many useful references on these subjects.

Directly related is the question of how to determine the surface area affected by an area brush. In the case of a spherical volume brush this amounts to finding the surface points inside a sphere centered at a particular surface point. In the pioneering paper on direct surface painting by Hanrahan and Haeberli [6] alternative approaches were discussed, of which the most intuitive is the 2D tangent-space brush. In this model the brush is represented by a planar polygon, which is projected onto the surface in the direction parallel to the surface normal at the brush center point. All surface points covered by this projection are affected by the brush color. A similar strategy has been pursued by Adams et al. [5], where an average normal across the surface area affected by the brush was used.

In the third category, methods have been developed to add the paint carried by the brush to the surface points. The transfer of color to a polygonal surface can be done by adding color to polygon vertices and shading the polygons during rendering [6], [7], or by using a surface parametrization and adding color to the points in the 2D parameter space, i.e., the texture domain. Vertex coloring restricts the color variation to the surface resolution, and it thus requires the surface to be tessellated high enough to be able to capture all painted details. For painting on point-sampled surfaces, Adams et al. [5] proposed a dynamic resampling scheme that can locally adapt the surface resolution to the brush resolution. Rischel et al. [8] employed uniform GPU-based subdivision to efficiently upsample a polygonal base surface, which, however, reflects any non-uniformity in the resolution of the initial surface.

If a surface parametrization is given, the user can paint directly on the 3D surface and the result of the painting is mapped onto the corresponding texture map [9]–[11]. This solution requires a low-distortion parametrization or a texture atlas [12] to yield a more uniform resolution of the texture map on the 3D surface, or an adaptive parametrization has to be computed locally to reproduce painted color details [13], [14].

To avoid the problems posed by vertex coloring and surface parametrization, octree textures were introduced by Benson and Davis [15] and DeBry et al. [16], and they were later realized on the GPU by Lefebvre et al. [17] and Lefohn et al. [18]. Octree textures make use of an adaptive yet regular sampling grid to store painted color details, hence avoiding parametrization issues at the expense of building and accessing an adaptive spatial data structure. In particular, the performance of surface coloring based on adaptive spatial data structures is limited by dependent memory access operations to determine affected surface samples. In Section 9 we will give quantitative evidence of these statements by comparing our approach to octrees on the CPU and the GPU.

An alternative spatial data structure for surface coloring was presented by Lefebvre and Hoppe [1], who employed perfect spatial hashing for building a unique mapping from surface points into the hash table into which the painting is directed. Perfect spatial hashing effectively minimizes the complexity of the data access operations, but it requires an exhaustive preprocess to build the mapping and can not guarantee in general that data access locality is kept.

4 Surface Representation

The surface coloring algorithms we present utilize a sample-based data structure for representing a surface and its attributes—the OFB—and an interface providing a set of operations on this structure. In an OFB, a surface is stored as a set of sampled surface points. Sampling is performed along three mutually orthogonal *sampling directions*, by projecting the surface orthographically along these directions into correspondingly aligned *sampling planes*. Every plane is discretized by a sampling grid, and each grid cell stores the distance to the sampling plane of the closest surface point projecting onto the cell center. In addition to this distance, the sampled surface color at this point is stored.

Since along one sampling direction up to d surface points can fall into the same cell (where d is the surface's depth complexity), up to d distances might have to be stored for each direction. Distances are sorted such that the i-th distance is the distance of the i-th closest point to the sampling plane. Every surface point is projected only once into the sampling plane with the smallest angle between the surface normal at that point and the grid's sampling direction. In this way, redundant sampling of the same point into multiple grids is avoided, and a nearly uniform sampling with a minimum and maximum sampling frequency of $1/(\sqrt{3} \cdot s)$ and $1/s$, respectively, across the surface is generated (where s is the size of an OFB cell). Figure 3 illustrates the sampling strategy used to generate an OFB.

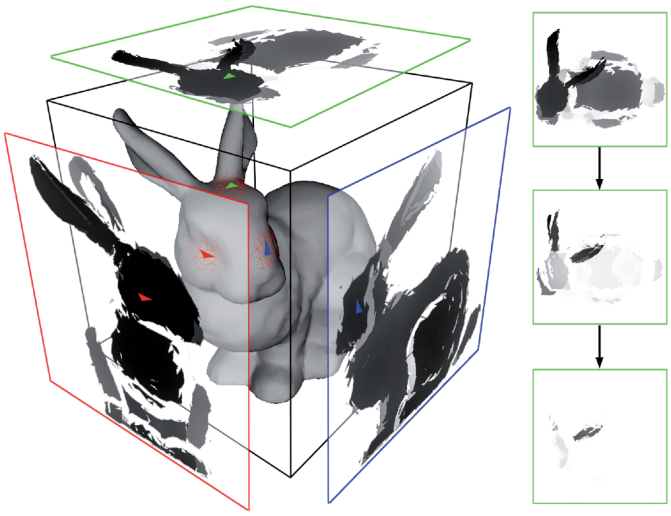

Fig. 3. Illustration of the OFB construction. Left: Surface points are projected along one of three mutually orthogonal sampling directions. Right: Surface points falling into the same grid cell are stored in multiple sampling grids.

Sampling the surface along a particular direction can be performed in many different ways, e.g., by using ray-casting or rasterization. In this work, the sampling is performed by rasterization on the GPU. In particular, we employ stencil routing [19], [20] in combination with a novel geometry shader algorithm to direct sampled surface points into the respective sampling grids (see Section 8). By using this approach the construction of an OFB can be performed at very high speed for surfaces at reasonable resolution.

4.1 OFB Interface

The OFB interface provides a method for locating a surface point in the OFB. The method takes as input a 3D position (x, y, z) in normalized object coordinates and tests whether a corresponding sample is stored in the OFB. To find this sample the point coordinate is projected into the OFB sampling plane most perpendicular to the point's normal. This generates for the respective sampling grid a 2D integer coordinate (u, v) and a distance, d, of the point to the sampling plane. If the sampling directions are aligned with the three coordinate axes the projection reduces to a component selection, i.e., in the z-direction $(u, v) = (\lfloor x \cdot S \rfloor, \lfloor y \cdot S \rfloor)$ and $d = z$, where S is the OFB grid size.

The distance d is now compared to all distances stored at index (u, v), and of all these values the index of the one closest to d within the interval $[d - s, d + s]$ is kept (with s being the cell size in the sampling grid). The grid identifier and the index are finally returned, and they can then be used to read a color value from the OFB or to write a color to it.

4.2 OFB Rendering

Rendering a surface using the colors that are stored in an OFB means to interpret the OFB as a texture consisting of several layers and fetching for every rendered surface point the color from this texture. This is realized by executing for every rendered point an OFB query as described to lookup the OFB sample closest to this point. The color at this sample is then read and used to modulate the point's appearance.

To support smooth color variations, the OFB interface provides distance-weighted color interpolation. If a surface is rendered at a resolution that is higher than the resolution of the OFB, for every surface point an OFB query is issued. In contrast to finding the closest sample, however, all samples within a radius of $\sqrt{3}$ times the size of a cell in the OFB grid are determined under all three projections. In each projection we also inspect the distance values in all grid cells adjacent to the cell (u, v). The color of a surface point is then computed from these samples by means of inverse distance weighting.

By using this interpolation scheme we can also generate an OFB mipmap hierarchy to resolve minification issues. Therefore, multiple OFBs at ever decreasing resolution are constructed, i.e., by subsequently reducing the resolution of the sampling grids about a factor of 2 in every dimension. Starting with the initial OFB at the finest resolution, the color of a sample at subsequent levels is computed by distance-based interpolation, with the color samples being fetched from the next finer level. In this way a stack of OFBs is generated, from which the appropriate resolution can be chosen during rendering (see Figure 4 for a comparison).

5 Surface Painting

Once the OFB structure has been constructed for the surface to be colored, the user starts painting with color or seeds particles leaving their color on the surface. In either case the surface point under the mouse cursor is used as center position.

In the most simple case, a spherical volume brush is used for painting. In this case, OFB samples closer to the brush center point than the selected sphere radius

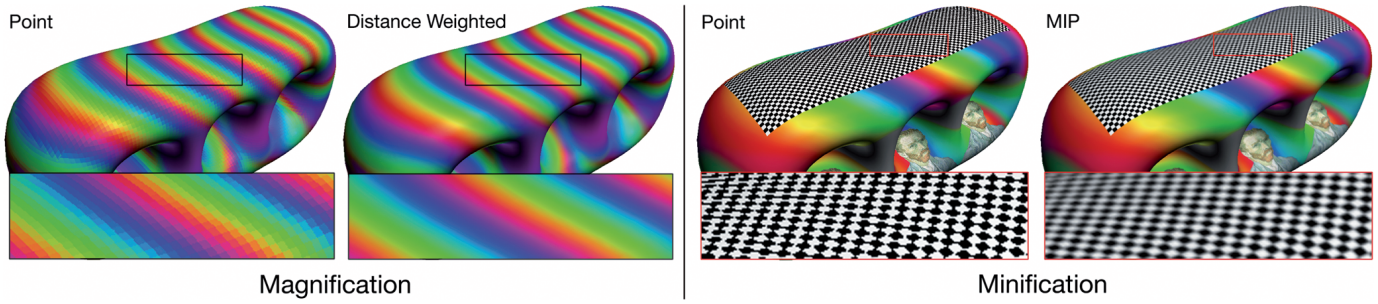

Fig. 4. Comparison of different OFB sampling filters to resolve minification and magnification issues.

are painted with the current paint color. The color of a sample at position P is updated according to

$$\mathrm{C}_P = \mathbf{lerp}(\mathrm{C}_P, \mathrm{C}_B, G), \tag{1}$$

where the brush shape G evaluates to

$$G = \begin{cases} 0, & \text{if } |P_C - P| > m \\ F\left(|P_C - P|\right), & \text{else.} \end{cases} \tag{2}$$

Here, P_C is the position of the center point, C_B is the brush color, and m is the support of a user-defined falloff function F, which is used to simulate smooth color fading with increasing distance to the center point.

It is clear that when a volume brush is used not every OFB sample should be tested for inclusion in the brush volume. Thus, a method for reducing the number of potential candidates to be tested is required. In vertex coloring, vertex topology can be considered to achieve this. Painting techniques working on point sampled geometry typically make use of a hierarchical spatial data structure such as a kD-tree to find all samples within a given distance to a center point. Approaches based on surface parametrization, on the other hand, can directly determine these candidates in the parameter domain if the parametrization is free of discontinuities. In our approach we exploit the fact that the OFB structure was built by sampling the surface along three mutually orthogonal directions. As a consequence, of all samples only those have to be tested whose projections along these directions fall into the regions covered by the projected bounding box of the brush volume.

On the CPU, the realization of this approach is straight forward by sequentially testing all OFB samples covered by the bounding box projections. To perform these tests for multiple brushes and samples within the brush volumes in parallel, and thus to achieve interactive painting even for many thousands of simultaneously used volume brushes, we now introduce a novel GPU method. This method exploits geometry shaders to efficiently determine all potential candidates and then performs the candidate tests in parallel in a pixel shader program.

A single vertex—positioned at the brush center point—is sent to the GPU and passed to the geometry shader. The geometry shader spawns three quadrilaterals from this vertex, each of which is aligned to one of the three sampling directions and rendered into all corresponding OFB sampling grids. The size of these quadrilateral is chosen according to the current extent of the brush volume. For every generated fragment, a pixel shader queries the sample in the OFB sampling grid, and it computes the distance of this sample to the brush center. Whenever a sample is closer to the center than the brush extent, the shader evaluates Equation 1 and writes the color into the OFB.

Interactive painting with a volume brush already provides an intuitive painting metaphor. In combination with a high-resolution OFB and by enabling the user to arbitrarily change the color and extend of the volume brush, finely detailed as well as large-scale paintings on surfaces can be created (see Figures 1a and 5). Especially in the first of the two figures it can be seen that our method is independent of the surface resolution. In this example, coloring was made at update times of less than 10ms on an upsampled version of the surface (53K triangles) consisting of 8.1 million samples.

On the other hand, as the volume brush model considers the Euclidean distance to the center point, surface points having a geodesic distance to the center that is larger than m may also be colored. This effect, which is often referred to as color leakage, is demonstrated in the closeup view in Figure 5. To overcome this limitation we are going to introduce an alternative brush model in Section 7.

Fig. 5. Fine color details are manually painted on the Stanford dragon using a spherical volume brush. The close-up shows color leakage introduced by a volume brush.

6 Surface Particles

In the following we introduce a new technique for surface coloring using so-called surface particles. Surface particles always stay on the surface and can be moved by external forces. They are seeded by the user at a particular position on the surface, and they can then be used to create a variety of different coloring effects such as color advection and convolution.

For a surface particle to move on the surface an external force field is required. Without loss of generality, we assume that this force field is given by a normalized vector field at the surface vertices. In the OFB construction this vector field is interpreted as an additional surface attribute, and it is sampled into the OFB in the same way as the surface's color. Before writing a vector field sample into the OFB, however, it is first projected into the local surface tangent plane. The tangent plane is computed from the interpolated surface normal, resulting in a smooth variation of the plane across the surface.

A surface particle—once released—steps along the surface and produces a sequence of surface points. At each of these points a volume brush is centered and used to spread color into the OFB. In addition to position, velocity, and lifetime, surface particles carry paint-specific information such as the color as well as the extent and transparency of the footprint they leave on the surface. Figure 6 demonstrates surface coloring via a surface particle that changes its color periodically from red over blue to green.

Fig. 6. A surface particle is released on the surface (white circle). The particle moves along a streamline in a vector field given on the surface and leaves color footprints along its path.

6.1 Particle Tracing on Surfaces

To trace a particle on the surface we compute its trajectory $P(u)$ parameterized over u, given the vector field $\vec{v}$ in 3D object coordinates and the particle's initial position (x, y, z) on the surface. This requires to solve the ordinary differential equation:

$$\frac{\partial P}{\partial u} = \vec{v}(P(u)) \text{ with initial condition } P(0) = (x, y, z)$$

To numerically solve this equation we employ classical Euler or Runge-Kutta integration using a fixed step size Δu. For a thorough overview of particle integration in vector fields let us refer to the report by Post et al. [21].

In principle, it is clear how to perform particle tracing on polygonal surfaces consisting of triangles [22]–[25]. Particles are traced from edge to edge by projecting the vector field onto the triangle plane and performing the particle integration in this plane. Although this approach can be realized in a straight forward way on the CPU, it imposes severe limitations on the number of particles that can be moved at interactive rates. Specifically, our tests have shown that not more than 10K particles per second can be integrated in one step on a triangle surface of reasonable resolution. As we will show in the remainder of this paper, however, some of the interactive coloring techniques we propose require up to a million particles to be moved per second, making a CPU solution impractical.

The GPU implementation of particle tracing on a triangle mesh, on the other hand, yields a highly non-uniform load in the parallel shader units performing the particle integration. While for a given Δu some units have to integrate over many triangles in one integration step, only one triangle might be considered by other units. On recent GPUs this results in execution stalls and thus in a significant loss of performance. This limitation can be avoided by tracing particles on the sample-based surface representation stored in an OFB. Since the OFB represents the surface at a nearly uniform resolution on a regular sampling grid, every unit performs the same number of memory access and numerical operations. In addition, by using the OFB as a structure for particle tracing, it can simultaneously be used to realize the color transfer to the surface.

Particle tracing on an OFB is performed by first locating the OFB sample closest to the current particle position as described in Section 4. To prevent a particle from leaving the sampled surface representation, its position is set to the position of this sample. The index of this sample is used to read the respective vector field sample, along which the particle is then moved to its new position.

A step along the surface is performed by first projecting the current vector sample into the three sampling planes. This gives for each plane a 2D vector t_u, t_v in this plane. By using these vectors and the projections of the particle position into the respective grids, we can now determine the grid cells in each grid into which the particle might be entering when making a step Δu that is equal to the cell size. In all of these cells we determine the sample closest to the new particle position, and we set the new particle position to the position of this sample.

Overall, given the index I of the start sample $P(I)$ in the OFB, in every iteration the following steps are performed (for the sake of simplicity we sketch the implementation of an Euler scheme here):

- **Vector lookup**: The vector sample $\vec{v}$ is read from the index I in the OFB.
- **Projection**: $\vec{v}$ is projected into the three sampling grids.
- **Integration**: Discrete cell traversal in the sampling grids along the projected vector sample and closest point location in the traversed cells yields the index I' of the OFB sample closest to P.
- **Update**: I is set to I' and the new particle position is set to $P(I)$.

It is clear that the accuracy of the proposed particle tracing method is limited due to the sample-based surface representation that is used. Since the particle positions are restricted to the OFB samples they will in general not accurately follow the characteristic lines in a given vector field. Due to this reason, we do not consider the method to be suitable for scientific vector field visualization. However, as we will show in the remainder of this paper, streamlines at high fidelity can be reconstructed by means of our method due to the extreme OFB resolution that can be used.

6.2 Particle-based Coloring

Surface particles released by the user carry paint, and they apply this paint to the OFB samples they run across. This is realized by automatically placing in every integration step and for every particle a paint brush at the particle position, and by transferring the brush paint to the OFB as described in Section 5.

Fig. 7. Ten thousand particles move along streamlines in a vector field given on the surface and leave color footprints on it. On our target architecture, particle advection and coloring takes less than 15 millisecond.

Particle-based coloring along the characteristic lines in a vector field designed on a polygonal surface is shown in Figure 7. In this example, 10K particles were simultaneously traced on the surface, each of them spreading a spherical color footprint to the surface. A more detailed analysis of this example is given in Section 9. Despite the huge amount of particles used, particle advection and coloring was performed at 80 fps on our target GPU.

6.2.1 Particle Chaining

By using chains of particles we can realize advanced brush shapes which would be difficult to realize otherwise. As an example we show how to simulate a brush shape that aligns with a given vector field and reduces its extend such as to simulate a tail (see Figure 8).

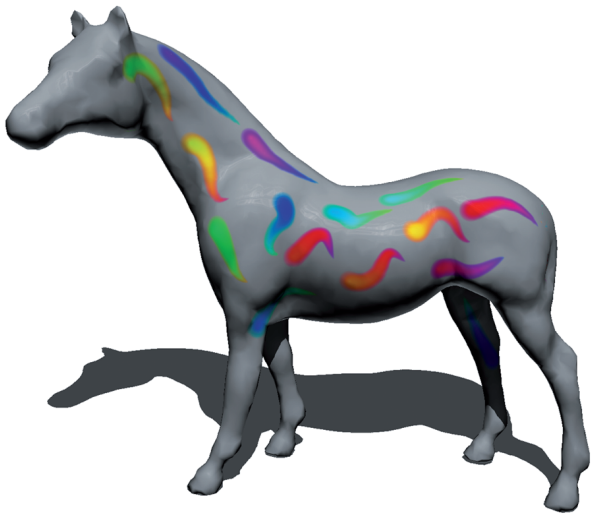

Fig. 8. Moving color spots with a tail are simulated by chains of surface particles. Two spots of different radius and color are simulated on top of each other.

In this example, at each point of a randomly selected set of points a sequence of particles is seeded consecutively. Each particle is assigned a counter which indicates its position in this sequence. All particles overwrite the color already stored in the OFB using an equally sized volume brush, but within the brush volume the brush color is faded out at ever faster decrease with increasing particle counter. In this way, the later a particle was released the smaller is the extend of its color footprint. By moving all particles along the vector field direction, the impression of moving particles with a tail is simulated.

6.2.2 Color Smearing

A smearing effect can be realized by letting a particle at position $P(u)$ query the OFB color at the previous particle position $P(u - \Delta u)$ along the particle trajectory, and by combining this color with its own color via a blending function S. This function controls the amount of smear, i.e., how much color from the previous position is carried over to the current position. The color of a

sample within the brush volume of a particle at position $P(u)$ is then computed as

$$\mathrm{C}_{P(u)} = \mathbf{lerp}(\mathrm{C}_{P(u)}, S\left(\mathrm{C}_B, \mathrm{C}_{P(u-\Delta u)}\right), G), \qquad (3)$$

with $\mathrm{C}_{P(u-\Delta u)}$ and C_B being the color at position $P(u-\Delta u)$ and the brush color, respectively. In case of a spherical paint brush, G simulates a radial falloff from the brush center such as to fade out the smeared color with increasing distance to the brush center. In Figure 9 this effect is demonstrated using an initial stripe-like color distribution in the OFB, and by setting C_B to $C_{P(u)}$ at every particle position. As can be seen, due to the linear interpolation between the current and the smeared color, the smear is fading out smoothly once a particle enters a region of different color.

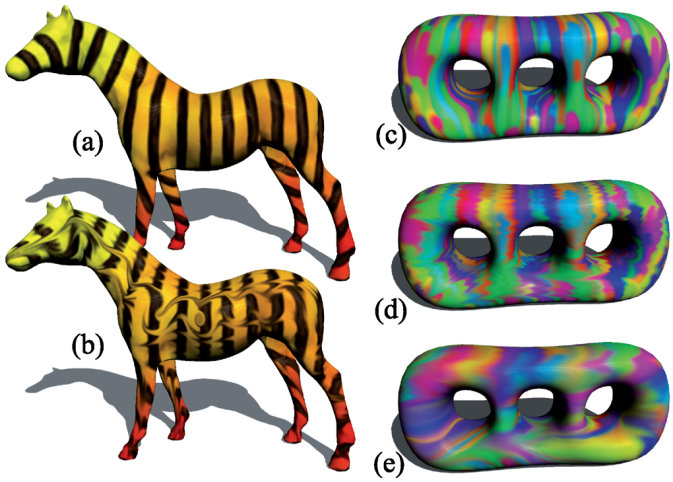

Fig. 9. Left: stripes of the "zebra" (a) are smeared by moving surface particles (b). Right: the color distribution on the triple donut (c) is smeared by a random vector field. The amount of smear is increased from (d) to (e).

6.2.3 Color Convolution

In the previous section we have shown how to use moving particles to create color traces that follow a given vector field. In contrast to this, color convolution alters a given color distribution in such a way as to show the directional information in the vector field. This method is commonly referred to as Line Integral Convolution (LIC) [23], [26], and it has been used in a number of approaches to interactively visualize vector fields given on a surface [27]–[29]. These approaches, however, distinguish significantly from ours in that they operate in image space on the visible surface points and do not transfer any color to the surface. For our intended application of surface coloring they are thus not suited.

Typically, LIC works by smearing a random noise intensity distribution along the characteristic lines in a vector field, which results in high intensity correlation along these lines as shown in Figure 10. Mathematically, this can be posed as the convolution of a color function C and a convolution kernel k along the characteristic lines:

$$Color = \frac{\int_{t=-L}^{L} C(P(p,t)) \cdot k(t)\, dt}{\int_{t=-L}^{L} k(t)\, dt} \qquad (4)$$

In our setting, a line integral convolution is computed at every OFB sample and the output values are written back into the OFB. The convolution is realized by spawning at every sample two surface particles, of which one is moved for some distance along its trajectory and the other one moves the same distance in the reversed vector field. While moving along the surface, the particles read the color from the OFB at the current position and weight this color with the kernel function. The accumulated color values from both particles are finally combined to yield the output value.

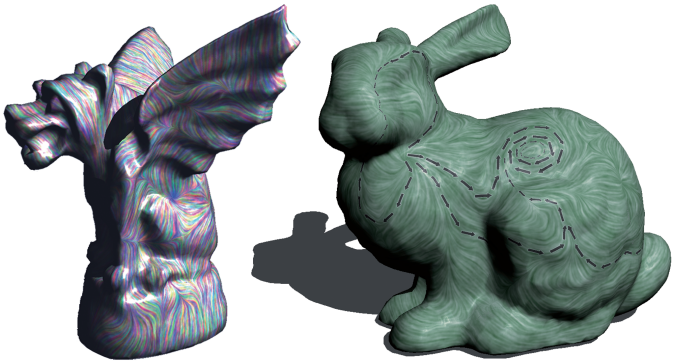

Fig. 10. Sample-based LIC on polygonal meshes. Since particles are traced on a 3D surface representation across silhouettes and edges, frame-to-frame coherence in animated scenes is guaranteed.

7 Surface-Aligned Brushes

As shown in Figure 5, a spherical brush can introduce color leakage because it considers the Euclidean distance of an OFB sample to the brush center point. Thus, surface samples having a geodesic distance to the center that is larger than the brush radius may also be colored. In the following we present an alternative brush model to overcome this limitation.

Ideally, the brush should be modeled as a deformable sheet that wraps around the surface and adds color to the points where it touches the surface. We call such a brush a surface-aligned brush, and we model it by a polygonal mesh consisting of surface particles connected via edges. Mapping the mesh onto a surface is done by tracing out the particles from the brush center point, which essentially corresponds to finding a local parametrization of the surface area surrounding this point.

Our method is thus similar in spirit to the patchinos and the exponential maps, which were introduced for decal painting by Pedersen [30] and Schmidt et al. [31], respectively. The patchinos, however, require a global parametrization of the base mesh. In contrast to exponential maps, on the other hand, we trace geodesics on

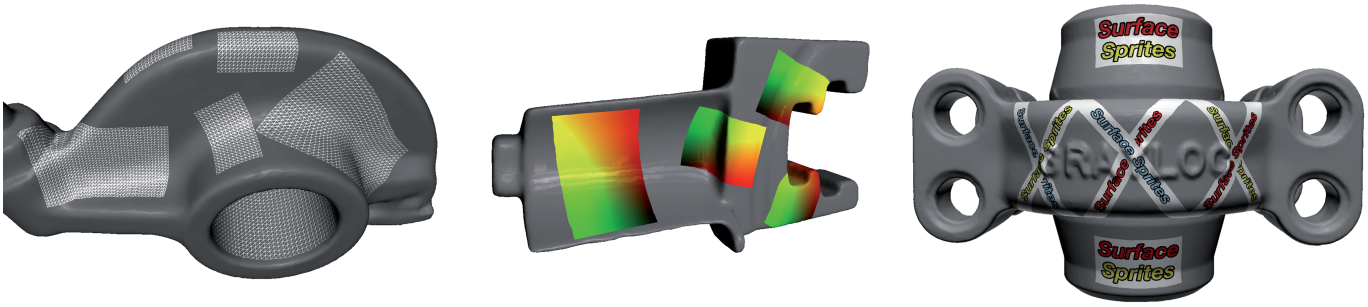

Fig. 11. From left to right: surface-aligned brush meshes rendered as wire-frame, shaded brush meshes with texture coordinates as color, and textured surface-aligned brushes.

the surface—or more precisely on a sampled version of the surface—instead of developing the surface to the tangent plane around a center point. Thus, the local parametrization we construct is completely independent of the underlying mesh resolution.

To align an areal brush on the surface, we first construct a local coordinate frame from a user-defined brush direction. It is build from the given direction vector (the tangent), the surface normal, and the vector perpendicular to both (the bi-normal). Constructing a local parametrization now starts by seeding two particles at the brush center point and tracing them along the surface. One of them is traced in the direction of the tangent vector and the other one is traced in the opposite direction. Since along these traces the tangent varies according to the change in the surface normal, it is corrected in every step as shown in Figure 12. In this way the trace wraps around the surface even in regions with high curvature.

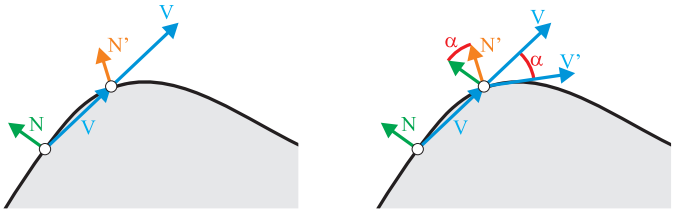

Fig. 12. The direction vector $\vec{v}$ is rotated α-degrees about the bi-normal, where α is the angle between the current normal N' and the normal N at the previous particle position in the plane perpendicular to the bi-normal.

After n steps, a polyline consisting of $(2 \cdot n)$ line segments is generated. From every particle on this line two new traces are started; one into the direction of the bi-normal and another one into the opposite direction. After m traces a 2D grid consisting of $(2 \cdot n + 1) \cdot (2 \cdot m + 1)$ particles has been laid out on the surface around the center point. Adjacent particles in the grid are finally connected to form a triangle mesh, yielding the local parametrization used to map the brush color to the surface.

The triangle mesh can now be texture mapped by specifying texture coordinates at the surface particles used to construct the mesh. The texture color can be transferred to the OFB by rendering the mesh and writing the color of every fragment into the OFB as described in 4. In Figure 11, a number of surface-aligned brushes have been used to bring color decals onto a surface. We have used rather large brush extends in this example to demonstrate the folding of the brush meshes along the surface.

8 OFB Construction

In the following we describe the method we use to efficiently construct an OFB on the GPU. Even though the surface coloring methods we have presented so far do not depend on the particular OFB construction method, a method that allows building the data structure at high speed is beneficial due to the following reason: In case that parts of the surface are not represented at a sufficient resolution in the OFB, such a method can be used to instantaneously rebuild the respective part of the OFB at the appropriate higher resolution. Although we have not yet implemented the surface coloring methods to work on multiple OFBs at different resolutions, this is a challenging future research direction that strongly relies on the availability of a fast method for OFB construction.

For single pass OFB construction on the GPU our method utilizes the geometry shader and the k-buffer introduced by Myers and Bavoil [20]. A k-buffer is a render target, i.e., a texture map, that can keep the contributions of up to k fragments per pixel instead of just one as in single-sample rendering. When rendering to a so-called multisampled texture target with multisample antialiasing being disabled, an incoming fragment is spread to all k multisamples of the respective destination pixel in the k-buffer. Since for each multisample a separate stencil mask is tested, stencil routing as proposed by Purcell et al. [19] can be used to direct an incoming fragment to a specific multisample. Stencil routing works by initializing the stencil mask of the i-th multisample with $i + 1$ (values 0 and 1 are used to identify empty samples and an overflow), and by letting a fragment pass the stencil test if the stencil mask is equal to 2. The stencil fail and pass operations are set to "decrementing", such that a stencil mask of 2 is consecutively obtained at all multisamples.

Via stencil routing up to k (=8 on our target graphics hardware) surface points seen under a pixel can be rendered simultaneously into one pixel of a supersampled render target. Since multiple render targets can be used and because an 8 bit stencil buffer is supported, surfaces with a depth complexity of up to 254 can be sampled in a single rendering pass. An OFB finally consists of three stacks of multisampled 2D-Texture slices, each of which stores distances of surface samples to the respective sampling planes. By using this approach, an OFB can be built at extreme resolution within a fraction of a second for typical surfaces used in computer graphics applications.

To efficiently sample the surface along three mutually orthogonal directions, we use the geometry shader (GS) available on DirectX 10 capable graphics hardware and under OpenGL as vendor specific extension. The GS's capability to direct its output to multiple render pipelines, each having its own depth, stencil and multiple color buffers, is employed to render a triangle into the appropriate sampling grid depending on its normal. The following setup is used on the GPU for that purpose, assuming the surface being represented as a triangle mesh with depth complexities d_x, d_y, d_z along the x, y, z-coordinate axes.

- **Pipeline Setup:** $T = \lceil \frac{d_x}{k} \rceil + \lceil \frac{d_y}{k} \rceil + \lceil \frac{d_z}{k} \rceil$ render pipelines are bound to the GS. To each of these pipelines k-times supersampled render targets are attached. Contiguous sets of $\lceil \frac{d_x}{k} \rceil$, $\lceil \frac{d_y}{k} \rceil$, and $\lceil \frac{d_z}{k} \rceil$ pipelines are used to perform the sampling along the x-, y-, and z-direction, respectively.
- **Geometry Shader Setup:** For every triangle, its face normal is computed by the GS and the triangle is directed into those pipelines who belong to the sampling direction with the smallest angle to the normal. Before triangles are sent to a pipeline, they are transformed according to the respective sampling direction, i.e. they are projected into a 2D sampling plane aligned perpendicular to this direction.
- **Pixel Shader Setup:** The pixel shader outputs the fragments depth as well as additional queried attributes into the supersampled render targets, and it decrements the stencil bits used to route fragments to the supersamples.

To store surface point positions and corresponding attributes, such as color or vector field samples, multiple OFBs are used. In DirectX 10, up to 8 buffers—with 4 32bit channels at most—can be bound as output targets to a pixelshader, enabling to capture up to 128 bytes of surface attribute data at once. Each OFB is initialized at startup or when the surface geometry is changed. The position OFB stores for every sample its distance to the corresponding sampling plane. Additional sample attributes are stored at corresponding positions in the respective other OFBs, i.e., a color OFB, a normal OFB, and a vector field OFB.

It should be finally noted that the OFB samples can be sorted with respect to their distance to the sampling plane [20]. Especially for objects having large depth complexity this can significantly improve the complexity of the OFB read-operation, from linear to logarithmic in the surface's depth complexity.

9 Results and Discussion

To validate the efficiency and quality of our approaches for surface coloring, we have used these approaches to interactively create finely detailed color structures on different triangular surfaces. Timings were performed on a 2.4 GHz Core 2 Duo processor and an NVIDIA 8800GTX graphics card with 768 MB local video memory.

In all of our examples, the three orthogonal OFB sampling planes were aligned with the faces of the axis-aligned bounding box of the object. To store the surface samples and their colors, for the sampling direction perpendicular to the largest face of the bounding box a $2K \times 2K$ sampling grid was used. The same cell size was chosen for the remaining two sampling grids. All other surface properties like vector samples and normals were sampled at half this resolution. Rendering was done to a 1280×1024 viewport.

Full statistics for some of the colorings performed in this work are given in Table 1. The measurements have been carried out in the following ways:

- Figure 1(a): A spherical volume brush of varying extent and color was used to manually paint color and colored letterings on an adaptive triangle mesh. The measurements are for a brush with a radius equal to 25 OFB samples.
- Figure 7: 10K particles were simultaneously traced on an OFB. In every integration step, each particle spreads a spherical brush of an extend equal to 5 samples to the OFB. The timings include particle integration in the OFB and coloring.
- Figure 1(d): Surface-aligned brushes with different color footprints were manually painted. Each brush mesh was laid out on the surface using 41×41 surface particles. The grids were mapped with a texture storing the respective color pattern. The timings include the layout of the surface brush and the color transfer into the OFB.
- Figure 10 (left): The measurements show the computation of LIC on the OFB representation of the Gargoyle mesh. At every OFB sample two particles were released, one of them traveling forward and the other one backward in a synthetic vector field. 10 integration steps were carried out along either direction. At every sample a sample-sized brush was used for paint transfer.

For each of these examples, Table 1 shows the surface's polygon count [in thousand] (column "$\triangle$"), the depth complexity along the three sampling directions (column "DC"), the number of surface samples in the OFB [in

million] (column "MSmpls"), the time required to construct the OFB (column "OFB"), and the OFB update rate in the course of painting/coloring (column "Paint"). All measurements are for unsorted OFBs.

	△	DC	MSmpls	OFB	Paint
Face	52	5 4 5	8.1	27	0.5
Horse	10	7 3 5	6.3	12	12.4
Hand	11	8 4 4	7.7	13	4.7
Gargoyle	10	6 4 4	11.2 (3.2)	16 (6)	1214 (268)

TABLE 1
Timing statistics in milliseconds for different coloring operations on surfaces with different triangle counts (△) and depth complexities (DC). In all examples an unsorted OFB was used. The numbers in brackets are with respect to an OFB of half the resolution as described. Rendering the surfaces with the colors stored in the respective OFBs took less than 5 milliseconds.

The statistics highlight the extreme resolutions at which surfaces can be represented in an OFB. Visually this is indicated by the very fine color details on all of the used models. Especially at the joint between the fine and coarse tessellation in the face model, it can be seen that the size of the painted details is independent of the surface resolution. It can also be seen from the statistics that even for models with reasonable depth complexity the efficiency of building an OFB on the GPU allows for ad-hoc construction.

With regard to the performance of the coloring methods our analysis demonstrates that all of them are fast enough to be used in interactive applications. In the second example we see, that even for a depth complexity of 7 along one of the sampling directions an effective throughput of more than 800 thousand particles per second can be achieved. This throughput includes the advection of particles via one-step Euler integration and the color transfer to the OFB.

The last example shows that even for a number of particles as large as 11.2 millions the computation of 20 integration steps in the OFB can be performed in slightly more than 1 second. This measurement includes the transfer of the accumulated color values to the OFB. By reducing the OFB resolution according to a $1K$ x $1K$ sampling grid, update rates of 8 frames per second can be achieved. This performance even allows animating the surface LIC representation by computing the convolution in every frame but letting particles start at subsequent positions along the characteristic lines.

To compare the efficiency of the data access operations in a sorted and an unsorted OFB, we have pursued an experiment in which an ever increasing number of nested cubes was colorized. By consecutively increasing this number the depth complexity of the scene is growing equivalently. For each configuration we have carried out a number of OFB access operations using randomly selected point coordinates, and we have measured the performance of returning the closest OFB sample for a sorted and an unsorted OFB. In a sorted OFB, for a point the closest sample in the three sampling grids is determined by a binary search in the sorted sequence of samples in the respective OFB cells. In an unsorted OFB, a linear search in this sequence is performed. The results are illustrated in Figure 13.

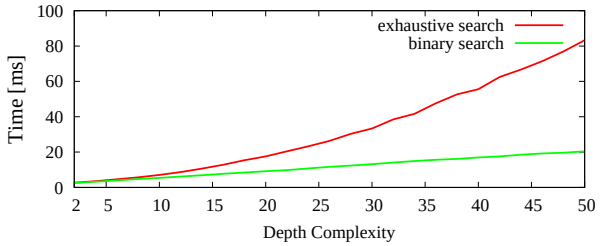

Fig. 13. Timing statistics for point location in sorted (green) and unsorted (red) OFBs with ever increasing depth complexity of the scene. For a depth complexity larger than 5 the logarithmic complexity of this operation in a sorted OFB pays off compared to the linear complexity of this operation in an unsorted OFB, even though the data access operations are now data dependent.

As can be seen, only for very small depth complexities the unsorted OFB yields comparable update rates, but quickly after a depth complexity of 5 the reduced number of data access operations pays off. Interestingly, this happens even though the data access operations are now data dependent, meaning that dependent texture fetch operations have to be performed on the GPU. Such operations typically stall the GPU processing pipeline since the time until the requested data item is available cannot be used for other computations. Apparently, however, the higher efficiency of non data-dependent access operations is completely amortized by the vastly reduced number of access operations in the sorted OFB.

9.1 Octree Comparison

In the following, we compare the performance of OFB-based surface coloring to octree-based surface coloring on the CPU and the GPU. For this purpose, we have implemented OFB- and octree-based coloring using spherical volume brushes on the CPU, and we have used the publicly available octree-textures implementation on the GPU by Lefebvre et al. [17]. The OFB was constructed as described in Section 9, and the resolution of the CPU octree was chosen accordingly. On the GPU, performance comparisons for 256^2, 512^2, and 1024^2 OFB sampling grids and correspondingly refined octrees have been carried out.

OFB-based painting on the CPU is performed in the same way as on the GPU, with the following minor differences: Firstly, the projection of the brush's bounding box along the three sampling directions is performed in software. Secondly, the samples covered by these projections are tested sequentially for inclusion in the spherical brush volume. The OFB itself is generated

	Construct (ms)		Paint (fps)		Memory (MB)	
	Octree (8/9/10)	OFB (256^2/512^2/1024^2)	Octree	OFB	Octree	OFB
Bunny	4934 / 15732 / 56445	3.0 / 4.5 / 7.0	25 / 12 / 3	360 / 343 / 309	4.25 / 16.1 / 52.4	6.75 / 27 / 108
Horse	4002 / 9818 / 31557	3.6 / 5.2 / 8.0	32 / 18 / 5	426 / 416 / 401	1.5 / 5.8 / 24.3	5.62 / 22.5 / 90

TABLE 2
Comparison between OFB- and octree-based surface painting on the GPU. The GPU octree implementation is publicly available at http://lefebvre.sylvain.free.fr/octreetex/. The table shows the performance (including painting *and* rendering) and memory requirements for octrees of depth 8,9, and 10, and respective OFBs with sampling grids of size 256^2, 512^2, and 1024^2. The first row gives the times for OFB and octree construction. The memory requirements of the octree implementation include only the color data stored at the leave nodes of the data structure. The additional memory required by the index structures could not be measured with the given implementation.

entirely on the GPU as described in Section 8, and the resulting sampling grids are downloaded to the CPU.

Our CPU octree implementation makes use of locational codes as proposed by Frisken and Perry [32] to support efficient point and region location for tree-based octree representations. For a given point, its coordinate values are first converted to x, y and z locational codes, and these codes are then used as branching patterns to iteratively locate the child nodes at every level of the tree. Each bit in a locational code indicates the branching pattern at the corresponding tree level, and the branching is performed until a leaf cell is reached. For querying so-called inradius-neighbors, i.e., the neighbors to a given node within a given search radius, first the level of the smallest enclosing cell in the tree is determined, and then the locational codes are used as described for every point contained in this region.

In Table 1 we have shown that it takes less than 13 milliseconds for 10K particles to simultaneously spread a color footprint with an extend equal to 5 samples to a high-resolution OFB. Note that this time includes the particle integration in the OFB. This means that more than 800 thousand of such paint operations can be performed per second on the GPU. Via region locations in the CPU octree we have achieved about 14 thousand of such operations per second, which indicates a performance gain of the OFB-based GPU approach of a factor of 57. By using the OFB representation on the CPU, we achieve a throughput of 92 thousand particles per second, which results in a performance gain of almost a factor of 6 over the CPU octree-based approach. In addition to showing the significant acceleration of surface coloring by using an OFB on the GPU, this analysis also demonstrates the improved algorithmic complexity of the data access operations in an OFB compared to an octree.

In Table 2, a detailed comparison between OFB- and octree-based surface painting on the GPU is given for two different data sets at different resolutions. In this comparison, manual painting using a brush radius equal to 5 OFB/octree leave nodes was performed. As indicated by Figure 14, visually the painting operations yield very similar results on both representations. OFB-based surface painting, however, performs up to two orders of magnitudes faster than octree-based painting, and it only needs about twice the memory of the octree implementation. Furthermore, while the OFB can be built at interactive rates, octree construction requires a significant portion of time.

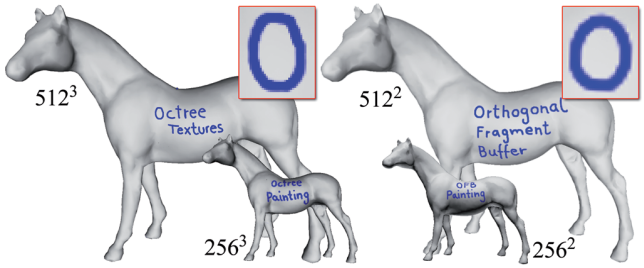

Fig. 14. Quality comparison between the GPU octree painter (left) and our OFB painter (right); in both systems linear texture filtering was used.

9.2 Applications

In the following we demonstrate two additional applications of the proposed coloring algorithms, which have not been described in the paper so far.

9.2.1 *Normal displacement*

All of the painting techniques described in this paper can also be used to paint into an OFB that stores additional surface normals. These normals are then used in the rendering of the surface to perturb initial surface normals. Figure 15, for example, demonstrates the use of a spherical volume brush to paint the illusion of bumps on the surface.

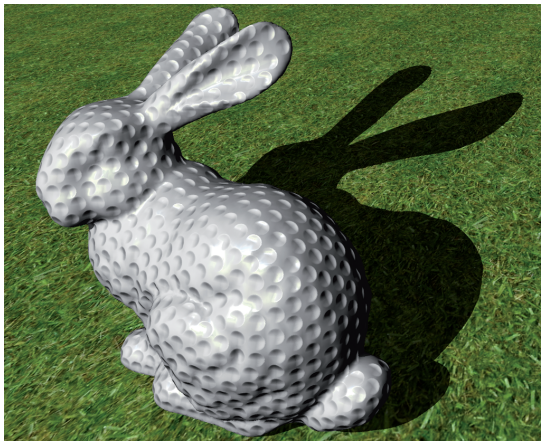

Fig. 15. A normal painted golf-ball bunny.

For every OFB sample within the support of the brush the vector from that point to the brush center point is computed. Both the normalized vector and its length, which is used as bump depth for parallax mapping, are stored. A user controlled scale factor allows controlling the depth of the bumps as well as its orientation, i.e., whether it faces away from the surface or towards it.

9.2.2 Dynamic Brushes

Instead of using one single 2D image to texture a surface-aligned brush, we can also loop through a stack of images from frame to frame. Internally this stack is realized as a 3D texture map. In this way we can realize an animated brush and we can also simulate the stroke that is produced by brushes of heterogeneously abrading material, such as charcoal. Therefore, in addition to the 2D texture coordinates assigned to each vertex of the brush surface, we simply use time or stroke length as the third coordinate into the 3D texture. This coordinate takes values from 1 to the number of 2D images used and periodically repeats this sequence while painting with the brush. To capture the behavior of the charcoal in Figure 16, the real image of a charcoal stroke on a rough material was scanned, cut into segments, and stacked into a 3D texture.

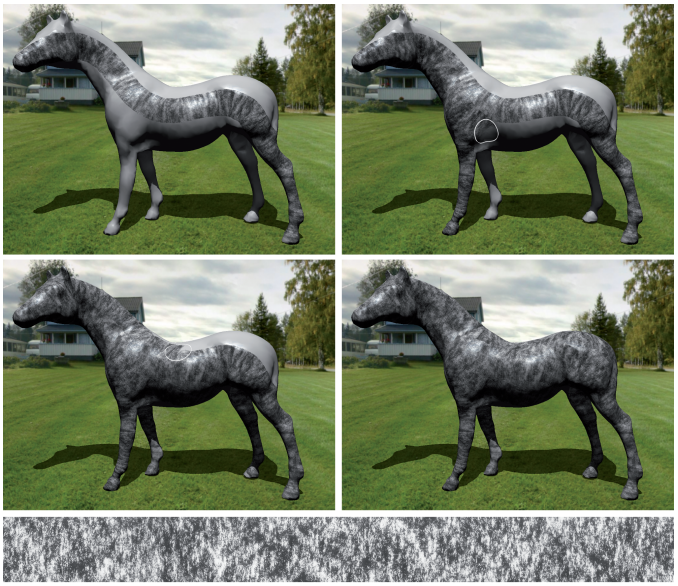

Fig. 16. Four charcoal paintings with a dynamic brush on the horse model. Underneath these images, the image of the scanned charcoal stroke that was used to create the dynamic brush is shown.

10 Conclusion and Future Work

In this paper we have introduced the Orthogonal Fragment Buffer—OFB—as a data structure for interactive surface coloring, and we have presented a number of different coloring techniques based on this data structure. Due to the efficiency and flexibility of these techniques, they are particularly suitable for artistic 3D content creation. The possibility to interactively add fine color details to a given surface—independent of the surface resolution and representation—distinguishes these techniques from previous ones. As an extension, it would be valuable to combine existing physics-based brush models with our techniques, including particular brush shapes and strokes as well as intuitive interfaces for particle seeding.

The limitations of our approach are twofold: Firstly, for models with high depth complexity it can require significant memory to store the OFB structure. Especially on the GPU the OFB resolution is limited due to the available texture memory. To alleviate this problem, we are currently looking into Direct3D 10.1 functionality to move data directly into S3TC compressed textures. More important is the observation that OFB structures are typically very sparse. Therefore, we will investigate alternative storage solutions based on adaptive texture maps, e.g., as proposed by Kraus and Ertl [33]. Because texture packing is very time consuming in general, we will investigate specialized packing strategies that exploit specific properties of the OFB. One such property is that the fill-rates of OFB layers along the sampling directions are monotonically decreasing, i.e., if a sample becomes empty in one layer it remains empty in all subsequent layers.

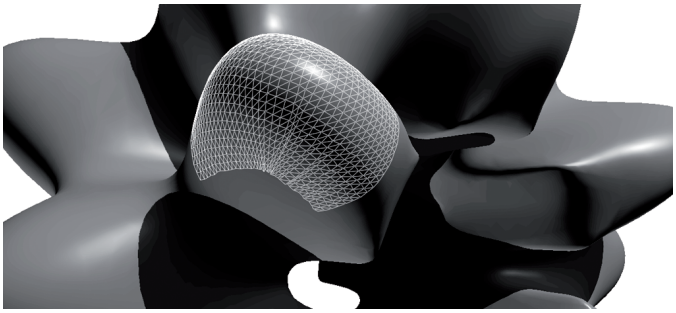

Fig. 17. A situation where particles traced out on the surface generate a folded brush mesh.

Secondly, we are aware that the proposed layout of surface-aligned brushes can produce distortions and even folds in highly curved regions (see Figure 17 for a demonstration of this problem). Similar to computing a local parametrization by the relaxation of surface samples as proposed by Adams et al. [5], in the future we will investigate the use of constrained mass-spring systems on the GPU for this purpose.

Finally, we will dedicate our research towards the finding of a render method that directly operates on the sample-based surface representation that is stored in an OFB, rather than mapping the OFB as a texture onto the polygonal surface representation. Similar to relief texture mapping [34], the surface geometry will then be replaced entirely by an image-based representation, i.e. the OFB, which is used in the rendering process.

One possible solution is to perform ray-casting on the regular 2D OFB sampling grids to determine primary-

ray surface intersections. Since this requires to test at every grid cell that is hit by a ray all samples within this cell, an additional acceleration structure has to be used. For instance, the OFB can be embedded into an adaptive space partition, where every part contains an OFB of only the surface part contained in this cell. In this way, the depth complexity per OFB, and thus the number of samples to be tested per OFB grid cell, can be reduced significantly. Another tempting alternative to reduce the number of ray/cell intersection tests during OFB traversal is to use so-called *safety radii*, which where introduced by Baboud et al. [35] for the efficient ray-casting of regular height fields.

ACKNOWLEDGMENTS

The authors wish to thank Leif Kobbelt and Peter Schroeder for providing some of the vector fields on surfaces shown in this paper. This work was made possible in part by the NIH/NCRR Center for Integrative Biomedical Computing, P41-RR12553-10.

REFERENCES

[1] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 579–588, 2006.

[2] D. Lischinski and A. Rappoport, "Image-Based Rendering for Non-Diffuse Synthetic Scenes," in *Proceedings, Ninth Eurographics Workshop on Rendering*, 1998, pp. 301–314.

[3] J. Shade, S. Gortler, L. wei He, and R. Szeliski, "Layered depth images," in *Proceedings of ACM SIGGRAPH 98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 1998, pp. 231–242.

[4] B. Baxter, V. Scheib, M. C. Lin, and D. Manocha, "Dab: Interactive haptic painting with 3D virtual brushes," in *Proceedings of ACM SIGGRAPH 2001: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001, pp. 461–468.

[5] B. Adams, M. Wicke, P. Dutre, M. Gross, M. Pauly, and M. Teschner, "Interactive 3D Painting on Point-Sampled Objects," in *Proceedings of the 2004 Eurographics Symposium on Point-Based Graphics (SPBG'04, Zurich, Switzerland, June 2–4, 2004*, 2004, pp. 57–66.

[6] P. Hanrahan and P. Haeberli, "Direct WYSIWYG Painting and Texturing on 3D Shapes," vol. 24,4, 1990, pp. 215–223.

[7] M. Agrawala, A. C. Beers, and M. Levoy, "3D painting on scanned surfaces," in *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, 1995, pp. 145–ff.

[8] T. Ritschel, M. Botsch, and S. Müller, "Multiresolution GPU Mesh Painting," in *Eurographics 2006 Short Papers*, 2006, pp. 17–20.

[9] D. Johnson, T. V. T. II, M. Kaplan, D. Nelson, and E. Cohen, "Painting textures with a haptic interface," *VR '99: Proceedings of the IEEE Virtual Reality 1999 Conference*, vol. 00, p. 282, 1999.

[10] A. D. Gregory, S. A. Ehmann, and M. C. Lin, "inTouch: Interactive multiresolution modeling and 3D painting with a haptic interface," in *VR '00: Proceedings of the IEEE Virtual Reality 2000 Conference*, 2000, p. 45.

[11] L. Kim, G. S. Sukhatme, and M. Desbrun, "Haptic editing of decoration and material properties," in *HAPTICS '03: Proceedings of the 11th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems (HAPTICS'03)*, 2003, p. 213.

[12] N. A. Carr and J. C. Hart, "Painting detail," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, 2004, pp. 845–852.

[13] T. Igarashi and D. Cosgrove, "Adaptive unwrapping for interactive texture painting," in *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, 2001, pp. 209–216.

[14] M. Zwicker, M. Pauly, O. Knoll, and M. Gross, "Pointshop 3D: An interactive system for point-based surface editing," in *Proceedings of ACM SIGGRAPH 2002: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002, pp. 322–329.

[15] D. Benson and J. Davis, "Octree textures," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 785–790, 2002.

[16] D. (grue) DeBry, J. Gibbs, D. D. Petty, and N. Robins, "Painting and rendering textures on unparameterized models," in *Proceedings of ACM SIGGRAPH 2002: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002, pp. 763–768.

[17] S. Lefebvre, S. Hornus, and F. Neyret, *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005, ch. Octree Textures on the GPU.

[18] A. Lefohn, J. M. Kniss, R. Strzodka, S. Sengupta, and J. D. Owens, "Glift: Generic, Efficient, Random-Access GPU Data Structures," *ACM Transactions on Graphics*, vol. 25, no. 1, pp. 60–99, Jan. 2006.

[19] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2003, pp. 41–50.

[20] K. Myers and L. Bavoil, "Stencil routed A-Buffer," in *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, 2007, p. 21.

[21] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch, "Feature extraction and visualisation of flow fields," in *Eurographics 2002 State of the Art Reports*, D. Fellner and R. Scopigno, Eds., Saarbrücken Germany, Sep. 2002, pp. 69–100.

[22] D. N. Kenwright and D. A. Lane, "Optimization of Time-Dependent Particle Tracing Using Tetrahedral Decomposition," in *VIS '95: Proceedings of the 6th conference on Visualization '95*, 1995, p. 321.

[23] D. Stalling and H.-C. Hege, "Fast and resolution independent line integral convolution," in *Proceedings of ACM SIGGRAPH 95: Proceedings of the 22th annual conference on Computer graphics and interactive techniques*, 1995, pp. 249–256.

[24] G. M. Nielson and I.-H. Jung, "Tools for Computing Tangent Curves for Linearly Varying Vector Fields over Tetrahedral Domains," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 4, pp. 360–372, 1999.

[25] P. Kipfer, F. Reck, and G. Greiner, "Local exact particle tracing on unstructured grids," *Computer Graphics Forum*, vol. 22, no. 2, pp. 133–142, 2003.

[26] B. Cabral and L. C. Leedom, "Imaging vector fields using line integral convolution," in *Proceedings of ACM SIGGRAPH 93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, 1993, pp. 263–270.

[27] J. J. van Wijk, "Image based flow visualization," in *Proceedings of ACM SIGGRAPH 2002: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002, pp. 745–754.

[28] R. S. Laramee, B. Jobard, and H. Hauser, "Image space based visualization of unsteady flow on surfaces," in *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, 2003, pp. 131–138.

[29] D. Weiskopf and T. Ertl, "A hybrid physical/device-space approach for spatio-temporally coherent interactive texture advection on curved surfaces," in *GI '04: Proceedings of the 2004 conference on Graphics interface*, 2004, pp. 263–270.

[30] H. K. Pedersen, "A framework for interactive texturing on curved surfaces," in *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 295–302.

[31] R. Schmidt, C. Grimm, and B. Wyvill, "Interactive decal compositing with discrete exponential maps," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 605–613, 2006.

[32] S. F. Frisken and R. Perry, "Simple and Efficient Traversal Methods for Quadtrees and Octrees," *Journal of Graphics Tools*, vol. 7, no. 3, pp. 1–11, 2002.

[33] M. Kraus and T. Ertl, "Adaptive texture maps," in *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2002, pp. 7–15.

[34] M. M. Oliveira, G. Bishop, and D. McAllister, "Relief texture mapping," in *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000, pp. 359–368.

[35] L. Baboud and X. Décoret, "Rendering geometry with relief textures," in *GI '06: Proceedings of Graphics Interface 2006*, 2006, pp. 195–201.

Kai Bürger is a PhD student at the computer graphics and visualization group headed by Professor Rüdiger Westermann at the Technische Universität München. There, he works on interactive techniques for computer graphics and visualization. Particularly, his research is focused on the development of GPU-friendly data structures and GPU-based solutions for real-time particle effects and interactive scientific visualization.

Jens Krüger is currently a research group leader at the DFKI Saarbrücken within the Cluster of Excellence MMCI. He also holds an adjunct faculty position at the SCI Institute at the University of Utah. In 2006 he received his PhD from the computer graphics and visualization group headed by Professor Rüdiger Westermann at the Technische Universität München. He has published papers on visualization, computer graphics and GPU programming at internationally recognized conferences like ACM SIGGRAPH or IEEE VISUALIZATION. In 2004 he received the ATI fellowship award, which honored him as an outstanding graduate student in areas related to computer graphics and graphics systems.

Rüdiger Westermann studied computer science at the Technical University Darmstadt, Germany. He pursued his Doctoral thesis on multiresolution techniques in volume rendering, and he received a PhD in computer science from the University of Dortmund, Germany. In 1999, he was a visiting professor at the University of Utah in Salt Lake City, and he became an assistant professor at the University of Stuttgart, Germany. In 2000, he was appointed an associate professor at the Technical University Aachen, Germany, where he was head of the Scientific Visualization and Imaging Group. In 2002, Westermann was appointed the chair of Computer Graphics and Visualization at the Technische Universität München. His research interests include general purpose computing on GPUs, interactive visualization, real-time rendering, hierarchical methods in scientific visualization, volume rendering, flow visualization and parallel graphics algorithms.