Efficient image reconstruction for point-based and line-based rendering

Ricardo Marroquim [a] Martin Kraus [b,*] Paulo Roma Cavalcanti [a]

[a] *COPPE-Sistemas, Universidade Federal do Rio de Janeiro, Cidade Universitária—Ilha do Fundão, Caixa Postal 68530, CEP 21945-970, Rio de Janeiro, RJ, Brasil*

[b] *Computer Graphics and Visualization Group, Technische Universität München, Boltzmannstrasse 3, 85748 Garching, Germany*

Abstract

We address the problem of an efficient image-space reconstruction of adaptively sampled scenes in the context of point-based and line-based graphics. The image-space reconstruction offers an advantageous time complexity compared to surface splatting techniques and, in fact, our improved GPU implementation performs significantly better than splatting implementations for large point-based models. We discuss the integration of elliptical Gaussian weights for enhanced image quality and generalize the image-space reconstruction to line segments. Furthermore, we present solutions for the efficient combination of points, lines, and polygons in a single image.

Key words: Point-based graphics, Point-based rendering, Line-based rendering, Pyramid algorithm

1 Introduction

Although there has been a steady increase of computer display resolutions for about three decades, there is still a large gap between the resolution achieved by today's displays and the resolution provided by mass-market printers. Thus, the resolution of desktop displays is likely to continue to grow as display

* Corresponding author. Tel.: +49 89 289 19482; fax: +49 89 289 19462
Email addresses: ricardo@lcg.ufrj.br (Ricardo Marroquim), krausma@in.tum.de (Martin Kraus), roma@lcg.ufrj.br (Paulo Roma Cavalcanti).

technology progresses. This will inevitably result in important challenges in hardware-accelerated graphics. In particular, Watson and Luebke [1] doubt that the bandwidth between GPUs (graphics processing units) and displays can grow at a sufficient pace without fundamental innovations in computer graphics. To this end, they and Dayal et al. [2] suggested "adaptive frameless rendering," which samples a scene by ray tracing it with an adaptive resolution in time and image space. The resulting set of samples is used for the image-space reconstruction of a considerably larger set of pixels of a display buffer.

However, today's GPUs are not designed to support real-time ray tracing; thus, it is interesting to consider alternative approaches to the challenge of adaptive sampling on GPUs. In particular, the adaptive sampling in image space can be replaced by an adaptive sampling in object space as performed by point-based renderers supporting dynamic level of detail [3–9]. Software-based image-space reconstruction in this context has been suggested by Grossman and Dally [10]. On the other hand, GPU-based reconstruction of point-based surfaces is usually performed by splatting techniques [3,4,11–15] although this approach provides a worse time complexity than image-space reconstruction [10]. In fact, surface splatting can be considered an object-space reconstruction of surfaces, which results in more pixel overdraw than image-space reconstruction for complex, screen-filling scenes.

For several years the main obstacle to efficient GPU implementations of image-space reconstruction techniques has been the efficient implementation of the pyramid algorithm, or—more specifically—the immediate read access to previously computed pixels without transferring texture image data between the GPU and the CPU. Although it was possible to avoid this problem as early as 2003, as shown by Krüger and Westermann [16], it took another three years until the first GPU implementations of the complete pyramid algorithm were published, for example by Strengert et al. [17].

Motivated by these publications, we designed a GPU implementation of an image-space reconstruction for point-based surface rendering [18], which achieved a better rendering performance than GPU-based splatting even without extensive optimizations. In Section 3 we discuss an improved algorithm, which provides a better rendering quality by integrating elliptical Gaussian weights. Moreover, we optimized our implementation for the latest GPU generation and report results for the reconstruction of additional surface attributes in Section 6.

While the efficient image-space reconstruction of point-based surfaces is an important problem in itself, it is also part of a greater challenge. In fact, rendering complex scenes with dynamic level of detail benefits from several graphics primitives in addition to points. For example, (textured) triangles are sometimes preferable due to the design of today's GPUs [5–7], and lines are

particular useful for many kinds of plants [19,8]. Therefore, we extended our image-space reconstruction to lines—or more specifically, to "ribbons" and "tubes"—as presented in Section 4. Furthermore, we discuss the integration of polygons into our approach in Section 5.

With this work, we demonstrate the efficiency and utility of GPU-based image-space reconstruction for various graphics primitives; in particular, points, lines, and polygons. Moreover, we propose the combination of adaptive object-space sampling with image-space reconstruction as a promising approach to real-time rendering for very high display resolutions.

2 Related work

This section mentions only some selected publications related to point-rendering techniques. More comprehensive surveys of point-based graphics have been published by Kobbelt and Botsch [20] and Sainz et al. [21,22]. The recently published book on point-based graphics edited by Gross and Pfister [23] includes also the latest research results in point-based graphics.

As published by Csuri et al. [24], points have been used to model and render "soft" phenomena such as smoke since the 1970s. At the same time, points have also been employed to render surfaces as published by Forrest [25]. Catmull and Clark [26] have presented a subdivision scheme for B-spline patches, which is suitable for rendering surfaces by subdividing them to sub-pixel points. This concept was implemented in the form of "micropolygons" within the Reyes architecture published by Cook et al. [27]. The first in-depth discussion of points as graphics primitive was provided by Levoy and Whitted [28].

Surface interpolation by a pyramid algorithm was suggested by Burt [29] and employed in an algorithm by Gortler et al. [30]. Here we will refer to this method as "the pull-push algorithm" although the terms "pull-push" and "push-pull" are also being used in many other contexts. The method was adapted by Grossman and Dally [10] for an image-space reconstruction of undersampled point-based surfaces, i.e., in the case of large gaps between projected points. Since the pull-push algorithm is of complexity $O(m)$ for a viewport of m pixels [29,30] and the projection of n points to single pixels is of complexity $O(n)$, the complexity of the whole algorithm is $O(n+m)$ [10].

While Grossman and Dally separated the detection and filling of gaps, Popescu et al. [31] described a hardware-architecture that implements a more elaborate separation of visibility and reconstruction for image-based rendering. For point-based rendering, Pfister et al. [3] employed the pull-push algorithm to fill holes between splats instead of one-pixel projections of points.

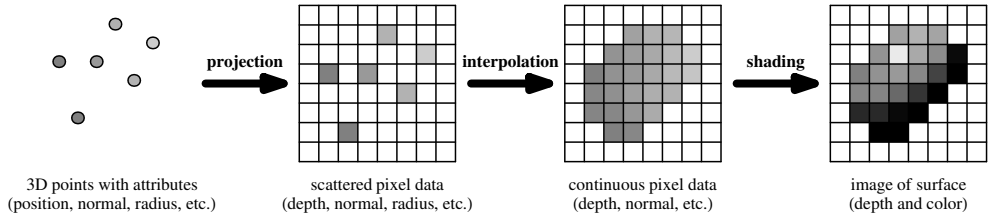


Fig. 1. Data flow in the proposed point-based surface rendering technique.

Subsequent research avoided these holes by splatting as suggested by Rusinkiewicz and Levoy [4]. This work presented a multiresolution point hierarchy supporting dynamic level of detail and was followed by several publications that integrated points and triangles in a single multiresolution hierarchy, for example, Chen and Nguyen [5], Cohen et al. [6], and Coconu and Hege [7]. Analogously, points and lines were integrated in one hierarchy by Deussen et al. [8].

Regarding the shape of splats, EWA (elliptical weighted average) splats as published by Zwicker et al. [32] are of particular interest. Hardware-accelerated EWA splatting was published by Ren et al. [11] and Guennebaud and Paulin [33]. Improvements of EWA splatting include perspective accurate splatting by Zwicker et al. [14], deferred splatting by Guennebaud et al. [9], and deferred Phong splatting by Botsch et al. [34]. Further improvements of this GPU-based approach were presented, for example, by Botsch et al. [15] and Guennebaud et al. [35].

Most of these techniques consist of a visibility pass to compute a depth map, an attribute pass to blend colors, normals, and other attributes, a normalization pass to finalize the interpolation of attributes, and a deferred shading pass. While this approach achieves a high rendering performance even for high-quality images, the efficiency of this GPU-based method is not optimal since the two object-order passes, namely the visibility pass and the attribute pass, have to process all displayed points, i.e., the time complexity of both passes is $O(n \times \bar{a})$ for n splats, each covering on average $\bar{a}$ pixels. Moreover, the two object-order passes result in a rather large constant for the dependency on n. On the other hand, the worst-case time complexity of the normalization pass and also of the shading pass (assuming local shading operations) is $O(m)$ for a viewport consisting of m pixels, i.e., it is independent of the number of projected splats. Thus, the total complexity of most splatting algorithms is $O(n \times \bar{a} + m)$.

Apart from these multipass splatting methods, several alternative point-rendering techniques have been presented; in particular, a "single-pass" splatting technique [36,37], dedicated splatting hardware [38], and ray tracing of points [39–41]. The "single-pass" splatting technique proposed by Zhang and Pajarola [36,37] has to compute groups of non-overlapping splats (in object space) and is therefore less suitable for large, dynamic point sets. While the dedicated

hardware published by Weyrich et al. [38] achieves promising performance results, the prototypical hardware cannot provide the rendering performance of splatting techniques implemented on today's GPUs. Ray tracing of points was first published by Schaufler and Jensen [39] and led to work by Adamson and Alexa [40], and Wald and Seidel [41]. The approach is particularly advantageous for many global illumination effects and for rendering of very large point models, which do not fit into graphics memory.

In spite of its linear complexity, there are not many hardware-accelerated implementations of the pull-push algorithm [30] because it requires a logarithmic number of switches of the render target. These switches have been a major bottleneck in the past; therefore, Lefebvre et al. [42] tried to avoid them in their approximation of the pull-push algorithm at the cost of a worse time complexity and reduced interpolation quality. However, a GPU-based implementation of the pull-push algorithm by means of the OpenGL extension for framebuffer objects performs extremely efficient as demonstrated by Strengert et al. [17]. This resulted in our application of a GPU-based pull-push interpolation to point-based rendering [18]. An improved variant of this method is presented in Section 3. Independently, Schnabel et al. [43] published a related rendering method using circular splats for compressed point clouds.

While points have been established as first-class graphics primitives, line segments (or "lines" for short) are less commonly employed. Line-based surface rendering was suggested by Wong et al. [44]; however, the lack of hardware support for wide, anti-aliased line segments is likely to impede its success. On the other hand, hardware-supported line-based rendering has been successfully employed by Deussen et al. [8] for the rendering of thin structures such as blades of grass or branches of trees as well as stream ribbons and thin stream lines in flow visualization by Zöckler et al. [45] and Mallo et al. [46]. A considerably improved image quality can be achieved by the use of proxy geometry instead of line strips as demonstrated by Stoll et al. [47] for quad strips and Merhof et al. [48] for triangle strips.

3 Point rendering using image reconstruction

Figure 1 presents an overview of our point-based surface rendering technique, which was recently published [18]. The input data of our algorithm consists of an unordered set of three-dimensional points with attributes, which are projected to the viewport. However, in contrast to splatting techniques, only one single pixel is rasterized for each point. Details about the projection are discussed in Section 3.1. After projecting the points, a continuous surface is reconstructed from the resulting scattered pixels by means of a pull-push interpolation as discussed in Sections 3.2 and 3.3. Based on the resulting contin-

uous pixel data, the deferred shading of the surface is computed as described in Section 3.4.

3.1 Projection of points to single pixels

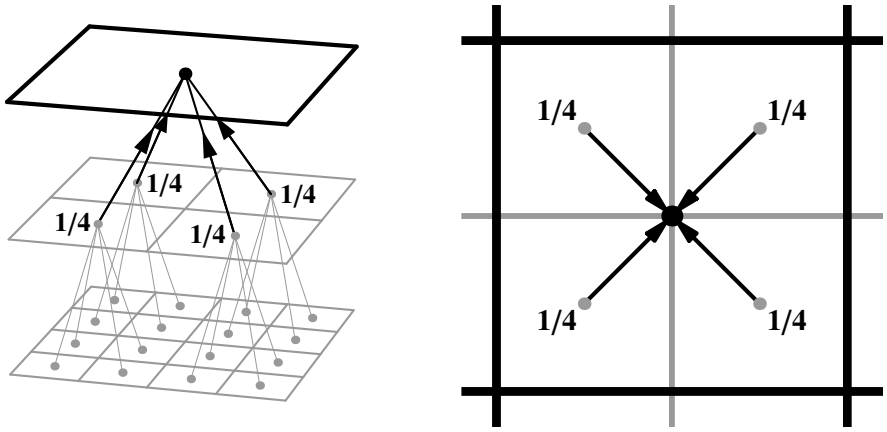
Points are projected to the viewport by a standard model-view matrix. In addition to viewport clipping, we also employ backface culling based on the local surface normal vector specified for each point. Apart from this normal vector, the only additional attribute required by our algorithm is a radius that specifies the extent of the point's influence. This radius corresponds to the splat size of splatting techniques and can be computed from the local sampling spacing. Since we are only concerned with point rendering in this work, we will not discuss the acquisition of points, their normal vectors, nor sampling spacings; instead we refer the reader to the literature on these problems [20,23]. For the same reason, hierarchical point representations and dynamic level-of-detail selection are not discussed in this work while they could be integrated in the point projection of our method.

Further point attributes are, for example, color and texture coordinates. For each point at most one pixel is rasterized; thus, the point's attributes are written to at most one pixel of the framebuffer. Since a depth buffer is employed for depth culling, each pixel stores the attributes of at most one point. Note that the point's attributes will usually consist of more than four components; thus, a hardware-based implementation requires support for sufficiently many multiple rendering targets.

Since at most one pixel is rasterized for each point, pixel overdraw is dramatically reduced in comparison to splatting techniques. In fact, this advantage was already noted by Grossman and Dally [10]. Another advantage of our approach is the limitation to one object-order pass, i.e., all points are processed only once. The requirement of two object-order passes, i.e., a visibility pass in addition to the attribute pass, is a major disadvantage of most hardware-accelerated splatting approaches as noted by Zhang and Pajarola [36,37] and Weyrich et al. [38].

3.2 Pull-push interpolation: pull phase

The pull-push algorithm consists of a pull phase and a subsequent push phase [30]. The former phase computes coarser levels of an image pyramid of the viewport image by reducing the pixel dimensions by a factor of two in each step. The push phase of our method employs this image pyramid to fill arbitrarily large gaps, i.e., to interpolate missing pixels and also to overwrite



(a) (b)

Fig. 2. Interpolation of pixel attributes in the pull phase.

pixels that are occluded by a surface as discussed in the next section.

In the pull phase, pyramid levels are computed in bottom-up order based on the viewport image containing projections of points. The attributes of a pixel of a coarser level are determined by averaging the corresponding four pixels of the finer pyramid level as illustrated in Figure 2. However, only pixels that specify valid data are included in the average. Whether a pixel is valid or not, is indicated by a binary flag per pixel. On the finest level, only the one-pixel projections of points are marked to specify valid data while all other pixels are marked invalid. When averaging four invalid pixels to compute a pixel of a coarser pyramid level, the new pixel is also marked invalid and is left to be computed during the push phase.

We limit the region of influence of each pixel by elliptical box-filters. If the center of a new pixel is outside a pixel's ellipse, the pixel is marked invalid for this new pixel. In our original algorithm [18], we employed these ellipses only in the push phase for inside/outside tests to limit the influence of points. In our improved variant, however, we have included this test also in the pull phase.

Each pixel's ellipse is computed by an orthogonal projection of a circle onto the view plane. The circle's orientation in object space is determined by the pixel's normal vector while its radius is set to the radius mentioned in Section 3.1. Thus, the ellipse's major axis is aligned perpendicularly to the 2D projection of the normal vector and its length is twice the radius; furthermore, the minor axis is parallel to the normal vector and its magnitude is the length of the major axis multiplied by the normal's z coordinate. While the center of an ellipse in the finest pyramid level is just the pixel's center, an additional 2D displacement vector is computed for each pixel to specify the ellipse's center in coarser pyramid levels. This vector is similar to the sample offset proposed by Popescu et al. [31].

After eliminating invalid pixels, a preliminary depth test is performed to also eliminate occluded pixels. To this end, each one-pixel projection is associated with a depth interval. The minimum depth of this interval is determined by the projected point's z coordinate while the maximum depth is computed by the minimum depth plus the radius of the point. A pixel is only used for averaging if its depth interval intersects the depth interval of the frontmost pixel, i.e., the pixel with the smallest minimum depth coordinate among the valid pixels, which potentially contribute to the average as depicted in Figure 2. Note that even though occluded pixels are removed from the interpolation of pixels of coarser levels, they are still present in the finer level. However, occluded pixels are recomputed in the following push phase as discussed in the next section.

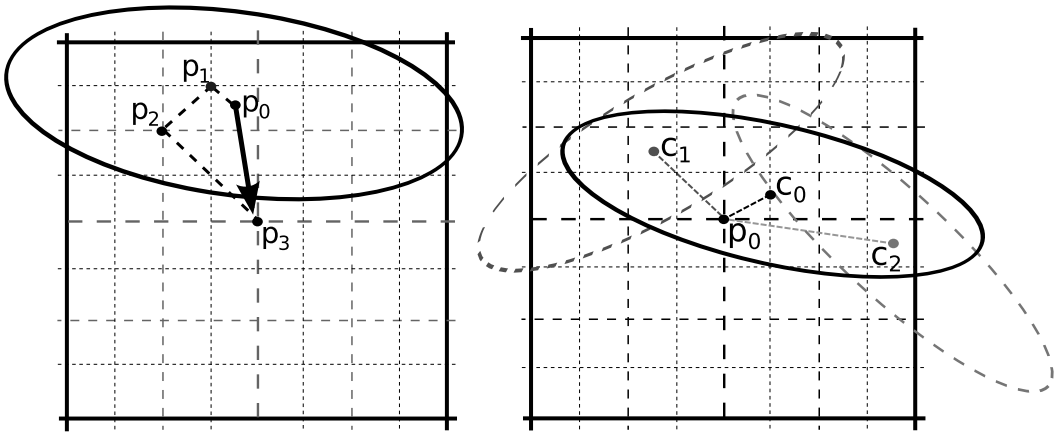
In general, the attributes of a new pixel of a coarser level are determined by averaging the valid, "unoccluded" pixels. However, the minimum and maximum depth values of a new pixel are set to the smallest and largest depth values of all contributing pixels, respectively. This guarantees that the depth interval of the new coarser pixel contains all intervals of the averaged pixels from the finer level.

Before averaging displacement vectors the difference vector from the corresponding pixel center to the center of the coarser pixel is added; the arrows in Figure 2b depict these vectors. This process guarantees that the coarser pixel maintains a reference to the exact position of the ellipse. An example of the evolution of the displacement vector during the pull phase is illustrated in Figure 3a. When a pixel of a coarser level is computed by averaging more than one valid pixel, the new normal vector, radius, and displacement vector define an ellipse that approximates two or more ellipses from the finer level (see Figure 3b).

3.3 Pull-push interpolation: push phase

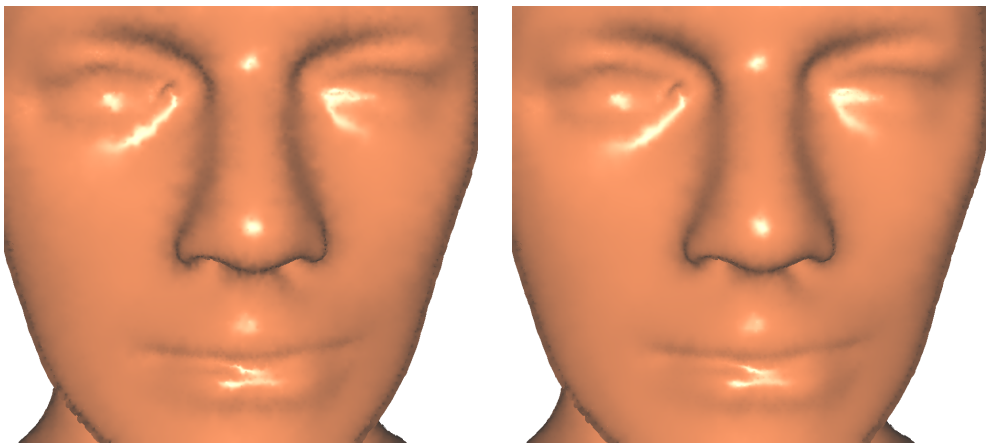
After the image pyramid has been built in bottom-up order in the pull phase, the push phase works in top-down order, i.e., from coarser to finer levels. In this push phase, only the attributes of invalid and "occluded" pixels are (re)computed. Here, a valid pixel is considered "occluded" if its minimum depth value is outside the depth interval of the corresponding pixel in the next coarser pyramid level. The attributes of four pixels of a coarser pyramid level are used to interpolate the attributes of a pixel of a finer level. Analogously to the pull phase, only pixels with valid attributes are included in the interpolation. If all four pixels are invalid, the new pixel is also invalid.

The interpolation scheme has to limit the influence of points according to the mentioned ellipses. To this end, our original algorithm [18] employed elliptical



(a) (b)

Fig. 3. (a) Displacement vector (thick arrow) for pixel p_3 computed after three steps of the pull phase. Pixel p_0 is the original projected sample at the center of the ellipse, p_1 is the center of the pixel at level 1, etc. The dotted lines represent the difference vectors summed at each iteration. (b) The attributes computed for pixel p_0 define a new ellipse (continuous line), with center at c_0, by averaging two coarser level ellipses (dotted lines) with centers at c_1 and c_2.

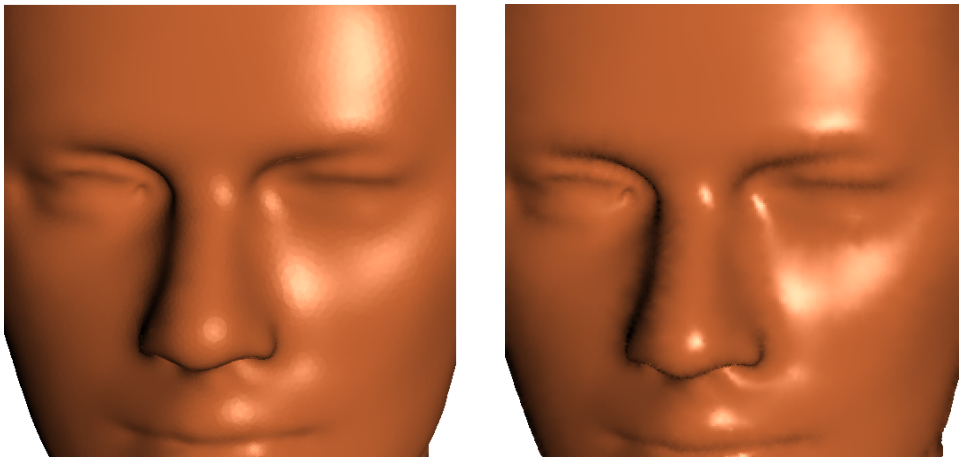


(a) (b)

Fig. 4. Comparison between our image-space reconstruction (a) without and (b) with Gaussian weights.

box-filters. In order to achieve a smoother interpolation, we replaced these elliptical box-filters by elliptical Gaussian kernels with finite support. This results in a weighted average of up to four pixels, which is then normalized by the sum of all weights. Figure 4 illustrates the difference between the pull-push interpolation without (Figure 4a) and with elliptical Gaussian weights (Figure 4b).

The push phase only recomputes invalid and occluded pixels; thus, the attributes of all other pixels are just copied from the image pyramid computed in the pull phase.



(a) (b)

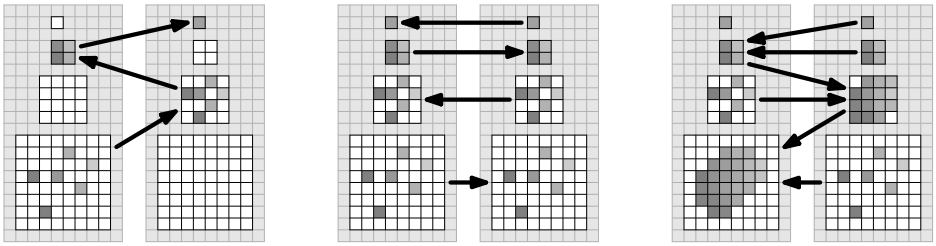
Fig. 5. Comparison between (a) non-deferred shading, and (b) deferred shading in our method.

(a) (b)

Fig. 6. Deferred shading with (a) constant material colors and (b) per-vertex diffuse colors.

3.4 Deferred shading

Once the push phase has been performed for all levels of the image pyramid, the finest level contains interpolated attributes for all pixels that are deter-



(a) (b) (c)

Fig. 7. Ping-pong rendering between two image buffers: (a) bottom-up pull phase, (b) copy phase, (c) top-down push phase.

mined to be within the silhouette of a reconstructed surface. For these pixels, Phong shading can be computed since the normal vectors have also been interpolated. Figure 5 compares non-deferred shading (in this case by a software splatting technique) in Figure 5a with the deferred shading by our method depicted in Figure 5b, which features sharper specular highlights.

Further per-point attributes such as texture coordinates or material colors can be included in the shading computation. Figure 6a depicts a point-based model with constant material colors, while Figure 6b shows the same model with interpolated per-point diffuse reflectivity colors.

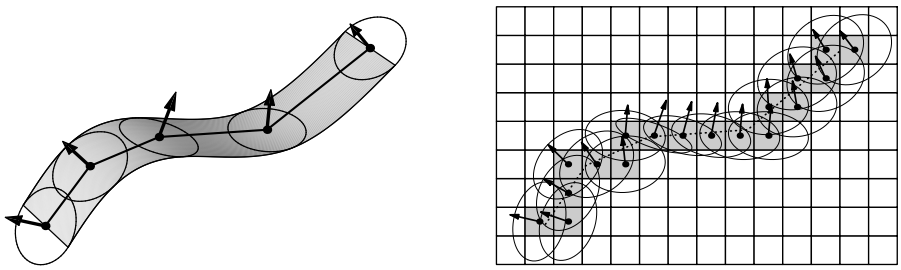
3.5 Ping-pong implementation on GPUs

Since our GPU implementation avoids simultaneous read and write access to image buffers, a ping-pong scheme between two image buffers is employed. Figure 7a illustrates this scheme for the pull phase while Figure 7c depicts the push phase. As the latter requires simultaneous read access to two levels of the pyramid, a third phase has to be included that copies the image data to complete the pyramid image in both buffers as illustrated in Figure 7b.

3.6 Discussion

There are some limitations of our approach to point-based surfaces rendering, which are discussed in this section. First of all, our method is not well suited for semi-transparent surfaces. In fact it is unclear how to robustly compute multiple depth layers using our approach. Since multiple depth layers are not available, edge anti-aliasing by alpha blending is impossible. Due to this limitation, the projection of points is performed without sub-pixel accuracy.

While the weighting of attributes by Gaussian kernels in the push phase of our method (see Section 3.3) improves the smooth interpolation of attributes,



(a) (b)

Fig. 8. (a) Representation of a ribbon by a polyline consisting of five vertices with normals and radii. (b) Illustration of the rasterization of a one-pixel-wide polyline corresponding to the centerline of a ribbon. Normals and radii are interpolated for each pixel covered by the polyline.

the smoothness of silhouettes is mainly determined by the finite elliptical support of these kernels. Thus, the image quality of silhouettes rendered by our approach is comparable to renderings obtained with non-anti-aliased, elliptical splats.

4 Line rendering using image reconstruction

In this section we extend the point rendering approach discussed in Section 3 to include the efficient rendering of line strips. More specifically, we distinguish between the rendering of ribbons, i.e., "flat line strips," and tubes, i.e., "cylindrical line strips." Ribbons and tubes differ not only in the employed illumination method but also in the way the local line width is computed as discussed in Sections 4.1 and 4.2. We also demonstrate applications of ribbons and tubes in plant rendering in Section 4.3 and discuss the efficiency of the proposed image-space reconstruction in comparison to previously published line rendering approaches in Section 4.4.

4.1 Ribbons

This section covers the reconstruction of "flat" ribbons of varying width and curvature as depicted in Figure 8a. For maximum efficiency, the image-space reconstruction of ribbons should be combined with the reconstruction of point-based surfaces discussed in Section 3. In other words, the pull-push interpolation should not only reconstruct smooth surfaces from scattered pixels but at the same time ribbons of finite width.

To this end, we represent each ribbon by a polyline that approximates the ribbon's centerline as depicted in Figure 8a. Each vertex of this polyline is

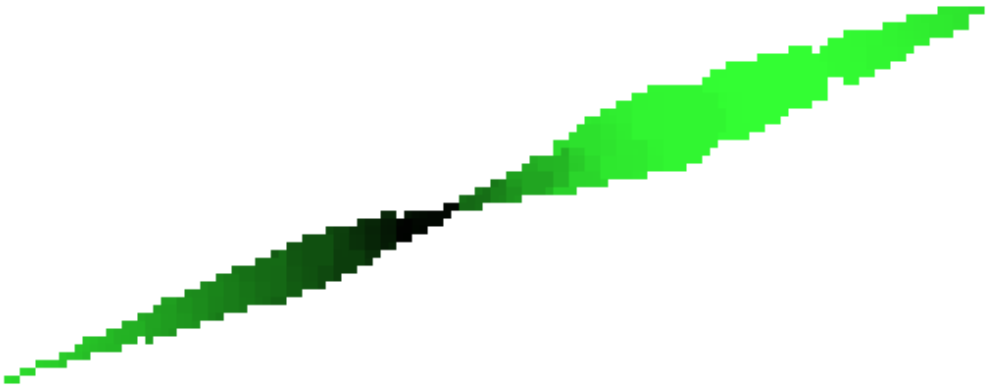
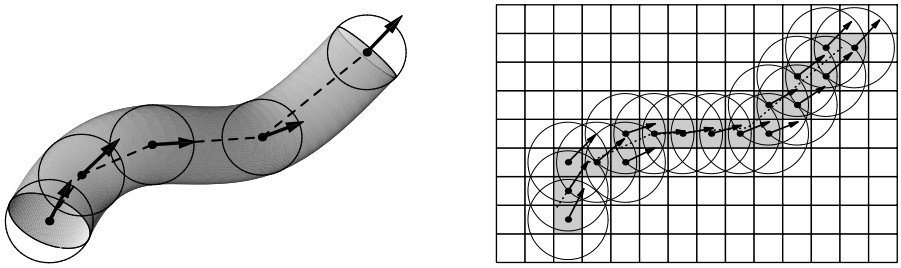


Fig. 9. Magnified rasterization of a twisted ribbon. The polyline consists of four vertices with per-vertex normals and radii as discussed in more detail in Section 4.3.

attributed with the local normal vector of the ribbon's surface and half its width. These attributes are interpolated for each fragment by the rasterization of the line strip as illustrated in Figure 8b. For best performance, hardware-supported line rasterization should be employed if possible. The minimum line width that guarantees a continuous set of pixels can be chosen for the rasterization since the pull-push interpolation will expand the rasterized polyline to the ribbon's width—just as single pixels are expanded to cover the area of larger splats as discussed in Section 3. Note that the rasterized line is not isotropically expanded; the width of the reconstructed ribbon rather depends on the local normal vector. This is most notable in the case of twisted ribbons as depicted in Figure 9.

Since each fragment that is generated by the rasterization of the line strip is rendered in a similar way as the points in Section 3, our method corresponds to modelling ribbons by as many splats as pixels are covered by the projection of the ribbon's centerline. However, only the vertices of a potentially coarse polyline have to be projected by the GPU. Moreover, the linear connectivity allows us to simplify line strips with very basic techniques; for example, a geometry shader could drop any number of inner vertices of a polyline depending on a specified level of detail.

There are also some minor differences between the reconstruction of surfaces and the reconstruction of ribbons. In contrast to the point-based rendering of surfaces, the line-based rendering of ribbons must not employ any kind of back face culling and has to compute correct lighting for front faces and back faces since usually both are visible. To allow the pull-push interpolation (and also the deferred shading) to distinguish between surfaces and ribbons, all pixels of all pyramid levels are attributed with an identifier specifying whether a pixel belongs to a surface, a ribbon, or a tube. The latter kind of line strips is discussed in the next section.



(a) (b)

Fig. 10. (a) Representation of a tube by a polyline. Each vertex is attributed with the tangent vector and the tube's radius. (b) Illustration of the rasterization of a tube's polyline. Tangent vectors and radii are interpolated linearly between vertices.

4.2 Tubes

The second kind of line primitives considered in this work are tubes, i.e., bended cylinders of varying width and curvature as illustrated in Figure 10a. Tubes require not only a different shading than the ribbons discussed in the previous section but they are also handled differently by the image-space reconstruction.

In analogy to ribbons, we represent a tube by a polyline that approximates the tube's centerline. Each vertex of this polyline is attributed with half the diameter of the tube and—in contrast to ribbons and surface points—the local tangent vector as illustrated in Figure 10a. Analogously to the case of ribbons, the polyline of a tube is rasterized as a one-pixel-wide line and the per-vertex attributes, i.e., radius and tangent vector, are linearly interpolated between vertices as illustrated in Figure 10b. In order to save memory, the tangent vector can be stored instead of the normal vector of ribbons and surfaces.

In contrast to ribbons and point-based surfaces, the pull-push interpolation for pixels covered by tubes always employs the normalized vector to the camera center as the surface normal vector. This is illustrated in Figure 10 by disks that are parallel to the view plane. Note that flat disks aligned in this way approximate the width of the projected tube very well in Figure 10a. This approach allows us to employ almost the same pull-push interpolation for surfaces, ribbons, and tubes since only the normal vectors are determined differently. Moreover, the pull-push interpolation can reconstruct all three primitives at the same time with minimal overhead compared to the reconstruction of only one kind of primitive.

The deferred shading of tubes can use any of the well-known methods for the illumination of lines [45,46,49,50]. The most basic approach for the diffuse illumination of a line by a single light source determines a surface normal $\mathbf{n}$ for diffuse lighting by the projection of the light vector $\mathbf{l}$ to the plane orthogonal

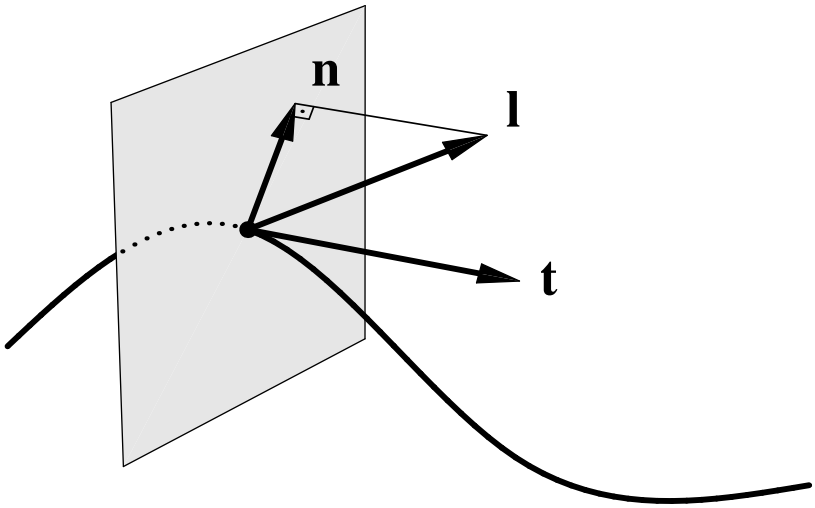
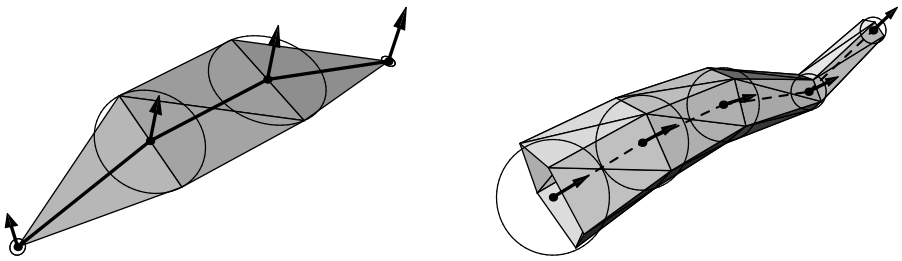


Fig. 11. Illumination of tubes: computation of a surface normal **n** for diffuse lighting of a line by projecting the light vector **l** to the plane orthogonal to the tangent vector **t**.



(a) (b)

Fig. 12. (a) Illustration of a leaf represented by four triangles and six vertices and the corresponding line-based model, i.e., a ribbon consisting of four vertices. (b) Illustration of a branch represented by triangles and the corresponding tube model consisting of only five vertices, which are attributed with radii and tangent vectors.

to the tangent vector **t** of the line as illustrated in Figure 11. Since the light vector is constant for directional lights (or can be computed from the position of a point light source and the three-dimensional fragment position), only the tangent vector is necessary to evaluate an approximative illumination of tubes. It should be emphasized that this approximation is only appropriate for thin projections of tubes; alternatives for higher levels of detail are discussed in Section 4.4.

4.3 Examples: leaves and branches

We illustrate the use of ribbons and tubes by demonstrating the conversion of a polygon model to a line-based model. In particular, we decided to employ the well-known model of an apple tree by Deussen and collaborators, which

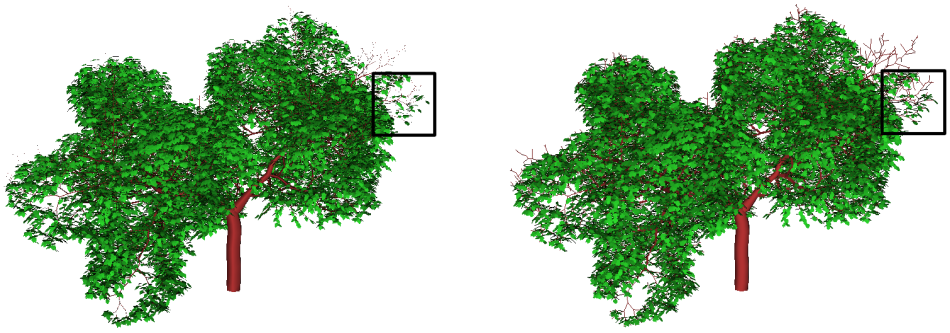
has appeared in several publications [8,19,51], including the front cover of the pbrt book [51]. Specifically, we employed the triangle model that was published on the CD accompanying the pbrt book. This model uses 200 496 triangles and 451 215 vertices to represent the leaves of the tree, while the bark is represented by a mesh consisting of 351 184 triangles and 371 583 vertices.

We converted the two triangle meshes separately since their structure is rather different. Each leaf of this particular model consists of four triangles and six vertices as illustrated in Figure 12a. We converted each leaf into a ribbon consisting of four vertices, where the end points are copies of two of the original vertices and the inner two points were determined by the averages of the two remaining pairs of original vertices. Normals were computed correspondingly while the radius of the end points was set to a value close to zero and the radii of the inner two vertices were determined by half the distance between the corresponding original vertices as depicted in Figure 12a. In this way we generated 150 372 ribbon segments and 200 496 vertices.

The bark of the triangle model represents a large tubular system of branches and was therefore converted to a set of tubes. The triangle mesh of each branch consists of a row of rings of (usually three) vertices, which are connected by elongated triangles. An illustration with rings of five vertices is depicted in Figure 12b. Due to this structure, we first heuristically determined clusters of vertices that are likely to form one of the rings of a branch. Each of these clusters corresponds to a vertex of our line-based model with a position that was determined by averaging the positions of all vertices in the corresponding cluster. Any pair of new vertices was connected by a tube segment if the corresponding pair of clusters was connected by an edge in the triangle model. The radius of each vertex was determined by the maximum distance of any of the original vertices of the cluster to the new vertex while the tangent direction was determined by the averaged direction of the attached tube segments. In this way we converted most of the bark to 63 887 segments and 69 604 vertices.

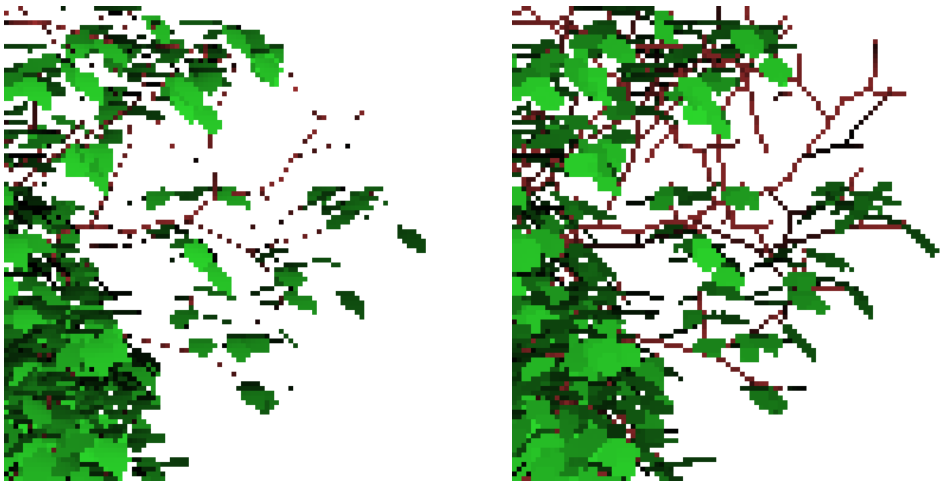
However, we did not convert the triangles representing the trunk. Instead we kept the 227 triangles and 681 vertices forming the bark of the thick trunk which is not well approximated by a thin tube. In fact, the illumination method for lines is not appropriate for the trunk unless the size of the projected model is very small. We could also convert the trunk to a point-based surface, but keeping the small triangulated surface offers an example for a hybrid model consisting of triangles and line segments. The implications of this combination are discussed in more detail in Section 5.

Figure 13 presents renderings of the original triangle mesh in Figure 13a and our line-based model consisting of ribbons, tubes, and some triangles representing the trunk in Figure 13b. With the computer system specified in Section 6 we achieve 31 frames per second for the rendering of the line-based



(a) (b)

Fig. 13. Comparison between (a) the original triangle model and (b) our rendering of a line-based model.



(a) (b)

Fig. 14. (a) Detail of the triangle-based rendering in Figure 13a. (b) Detail of the line-based rendering in Figure 13(b).

model while rendering the original triangle model achieved 28 frames per second.

In summary, the conversion of the triangle model of an apple tree to a line-based representation consisting of ribbons and tubes is rather straightforward in this particular case. Considering that the original triangle mesh provides only a coarse approximation to real leaves and branches, our line-based model is sufficiently accurate for the leaves and thin branches while it requires only about one third of the vertices.

Thus, line-based models appear to offer an attractive alternative to polygon-based models for low level-of-detail representations; in particular since we expect that the conversion of procedural plant models poses even less problems than the conversion of polygon models. Moreover, the simplification of line-based models is straightforward; thus, hierarchies of coarser approximations can be more easily computed than for polygon-based models, as discussed by

Deussen et al. [8].

4.4 Discussion

Image-space reconstruction for line-based rendering offers some important advantages in comparison to published line-based rendering approaches, which we discuss in more detail in this section. Specifically, we compare our method with the rendering of triangle meshes, line strips, and proxy geometry.

Representing thin structures, such as hair, fur, grass, twigs, etc., by triangle meshes requires more vertices than a line-based model of similar quality; moreover, rendering simplified versions is considerably more difficult for triangle models than for line-based models. Even though modern GPUs are designed to achieve their optimal performance for triangles, line rendering is also well supported, i.e., rendering the longest edge of a triangle is usually not slower than rendering the triangle. On the other hand, the rasterization of lines requires fewer fragments and the rendering of line-based models tends to project fewer vertices. Therefore, our method can provide an advantageous performance for large, screen-filling scenes since the pull-push interpolation imposes only a constant overhead. One important limitation of our approach to line-based rendering is the lack of anti-aliasing; thus, lines rendered by our method are at least one pixel wide. Therefore, thin structures will appear thicker if compared with the corresponding triangle rendering. This can be observed in the magnified detail of the triangle-based rendering depicted in Figure 14a in comparison to the line-based rendering in Figure 14b.

Deussen et al. [8] employ hardware-supported rendering of line strips for low level-of-detail representations of plant models. Since this approach is restricted to a uniform line width, it is only useful for very coarse projections of models. Moreover, rasterization with a uniform line width corresponds to the rendering of tubes as discussed in Section 4.2, which would require a line illumination model. However, Deussen et al. employ the per-vertex normals directly, which is only appropriate for ribbons. These, however, cannot be rasterized with a uniform line width. Thus, the fixed line width required by many graphics APIs severely limits the use of line strips for line-based rendering. In comparison to the rasterization of line strips, our pull-push interpolation imposes a constant overhead; therefore, it cannot achieve the same performance.

Tubes of finite radius can be efficiently rendered by employing proxy geometry, e.g., quad strips, in combination with per-fragment computations of ray-cylinder intersections as suggested by Stoll et al. [47], or a combination of triangle strips and point sprites as suggested by Merhof et al. [48]. While these approaches achieve a considerably higher image quality than our method, their

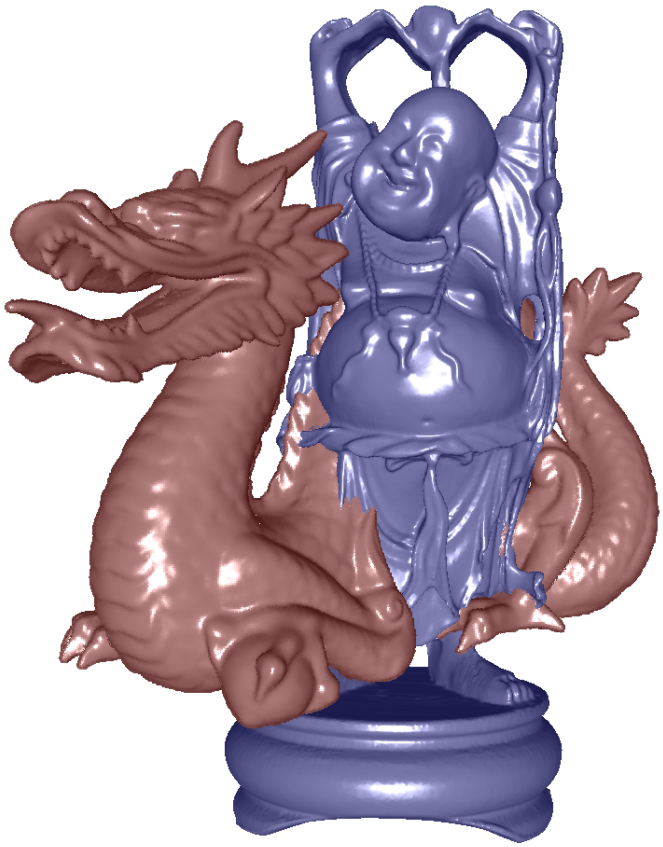


Fig. 15. Combined rendering of a point-based model (Dragon) and a triangle-based model (Buddha).

performance is—according to Stoll et al.—an order of magnitude worse than the rendering of shaded line strips, while our method imposes only a constant overhead. Merhof et al. reported a decrease in performance by a factor of 3.6 from illuminated line strips to their method for a particular scene. The programmable generation of geometry streams offered by recent GPUs is likely to improve the performance of these approaches, however, the number of vertices of the proxy geometry that have to be processed is not reduced and the fragment stage is also not affected; thus, the potential improvements are limited and the methods are likely to always perform significantly worse than the rendering of line strips.

Overall, the proposed image-space reconstruction of lines provides very high rendering performance for lines of finite width, in particular for large scenes and/or if the constant overhead imposed by the pull-push interpolation is amortized by the combined rendering with point-based surfaces as described in the next section. However, the limited image quality achieved by our method restricts it to thin lines or coarse renderings with low level of detail; thus, it is best suited to render coarse representations of line-based models within real-time renderers using dynamic level of detail.

5 Combining points, lines, and polygons

It is unlikely that one kind of graphics primitive is optimal for all elements of a scene; moreover, rendering approaches with dynamic level of detail may choose different representations of the same object using different kinds of primitives depending of the required level of detail. Thus, the combined rendering of polygon meshes, point-based surfaces, and line-based models in a single frame is an important scenario, which is discussed in the next section. Apart from the combined rendering of representations based on different primitives, these primitives can also be combined to represent a single object by a hybrid surface model as discussed in Section 5.2.

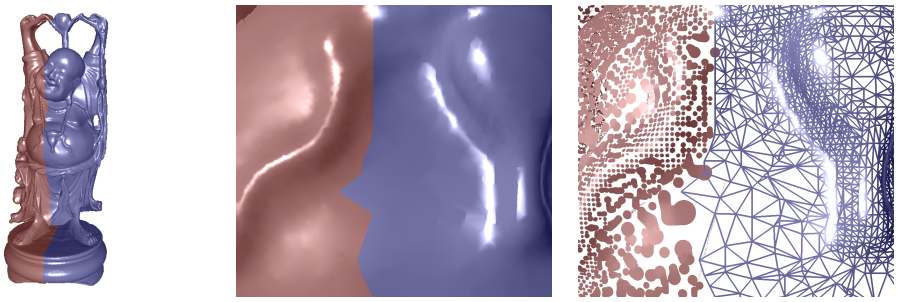
5.1 Combined rendering of different primitives

While the image-space reconstruction of line-based models proposed in Section 4 was designed to support the integration of points and lines in the same pull-push interpolation, the combination with polygons poses additional challenges. One solution would be to compute one RGBA image with depth data for polygon-based graphics and a second image for point-based and line-based graphics. The resulting images can then be combined according to their depth and alpha channels. While this approach offers some advantages for semitransparent objects, it requires additional memory and pixel processing.

Therefore, we propose an alternative approach, which integrates the rendering of polygons in the image-space reconstruction of surfaces and lines. To this end, polygons are rasterized into one frame buffer together with the single-pixel projections of points and the one-pixel-wide polylines of ribbons and tubes. However, the pixels covered by polygons are assigned a radius close to zero; thus, they are not expanded in the pull-push interpolation. This approach imposes no overhead on our system since the performance of the pull-push interpolation is not affected by the content of the frame buffer. An example is depicted in Figure 15.

5.2 Hybrid surface models

As described in the previous section, the fragments of polygons can be considered points with a very small radius of influence in order to integrate them smoothly in the pull-push interpolation. This concept can also be applied to construct models with a smooth spatial transition between a polygon-based part of a surface and a point-based part. To this end, the vertices of the boundary edges between the two parts have to be attributed with radii that are large



(a) (b) (c)

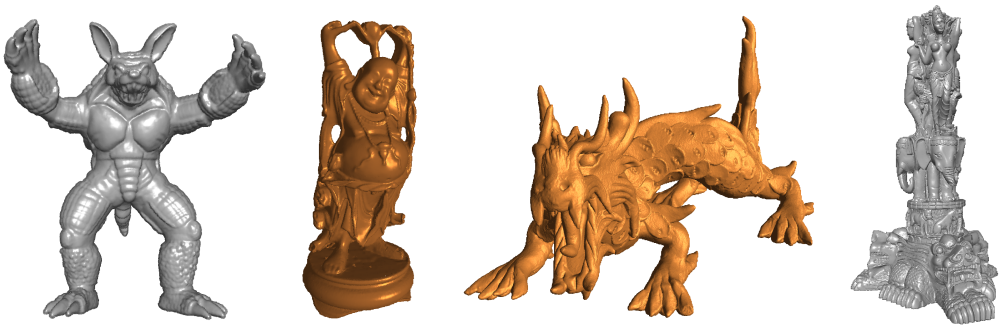
Fig. 16. A hybrid surface model: (a) The blue half consists of triangles, the remaining model is point-based. (b) Rendering of a detail and (c) illustration of the underlying primitives.

Table 1
Models and rendering performance. Each total rendering time per frame is followed in parentheses by the time for the point projection and the time for the pull-push interpolation (all rendering times are in milliseconds).

		without color buffer		with color buffer	
model	# points	fps	time per frame	fps	time per frame
Armadillo	173 K	89	11 ms (1.2 ms, 8.5 ms)	46	22 ms (1.5 ms, 18 ms)
Dragon	437 K	78	13 ms (2.2 ms, 8.9 ms)	44	23 ms (2.9 ms, 18 ms)
Happy Buddha	544 K	76	13 ms (2.6 ms, 8.8 ms)	42	24 ms (3.4 ms, 18 ms)
Asian Dragon	3610 K	36	28 ms (18 ms, 8.3 ms)	23	45 ms (25 ms, 17 ms)
Thai Statue	5000 K	29	35 ms (25 ms, 8.2 ms)	18	55 ms (34 ms, 18 ms)

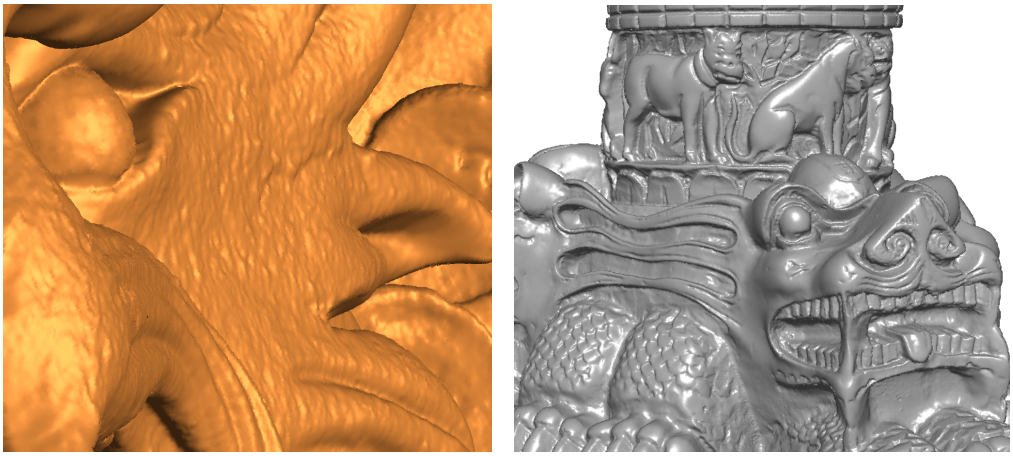
enough to let the rasterized pixels of the boundary edges blend smoothly with the single-pixel projections of the point-based continuation of the surface. Figure 16 illustrates a hybrid surface model consisting of points and triangles. This kind of combination occurs frequently when multiresolution hierarchies based on points and triangles [5–7] are employed.

If backface culling is deactivated and two-sided lighting is provided, ribbons can be combined analogously with points and polygons to form continuous surfaces. Tubes, however, pose additional problems since neither the different illumination models of lines and surfaces allow for a smooth transition nor is a smooth interpolation between tangent and normal vectors trivial.



(a) (b) (c) (d)

Fig. 17. Renderings with our method of the (a) Armadillo, (b) Happy Buddha, (c) Asian Dragon and (d) Thai Statue.



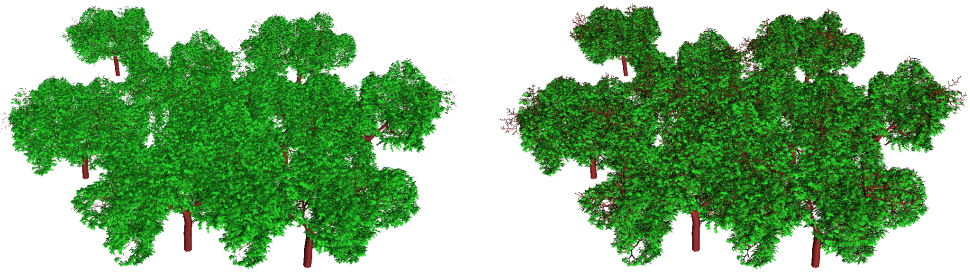
(a) (b)

Fig. 18. Renderings of details of the (a) Asian Dragon and (b) Thai Statue.

6 Results

We tested our algorithm on a GeForce 8800 GTS with 640 MB memory connected via PCI Express 16× to a Linux computer with an Intel Core Duo 6600 CPU (2.4 GHz), 2 GB RAM, and installed NVIDIA driver, version 169.04. The models were preprocessed to compute a normal vector and a radius of influence of each point. At runtime, all point attributes were transferred to the GPU by means of OpenGL vertex buffer objects. Image data was processed in an OpenGL framebuffer object with 16-bit floating-point RGBA image buffers and a 32-bit depth buffer with a viewport size of 1024×1024 pixels. All vertex and fragment shaders were implemented in the OpenGL shading language.

Rendering times for several point-based models are summarized in Table 1; exemplary renderings of these models are depicted in Figures 4, 17, and 18. The fourth column of Table 1 presents rendering times in milliseconds without interpolation of a surface color while the sixth column presents times including



(a) (b)

Fig. 19. Rendering of 10 trees based on (a) the original triangle model and (b) our line-based model.

the interpolation of a surface color. Apart from the total time per frame, we have also included two times in parentheses: the time required for the projection of points as described in Section 3.1 and for the pull-push interpolation discussed in Sections 3.2 and 3.3. As expected, these two operations require most of the rendering time while other operations, e.g., the deferred shading, are almost negligible.

Our measurements demonstrate that the rendering time for small models is dominated by the pull-push interpolation; i.e., by the viewport resolution as demonstrated in our previous work [18]. On the other hand, the rendering time for large models is dominated by the projection of points and therefore by the number of points. For large models without interpolation of surface color, our implementation renders the equivalent to about 130 M splats per second—including surface reconstruction and deferred shading. If the interpolation of surface colors is included, the rendering performance is reduced to about 90 M splats per second.

Our previous implementation using a GeForce 7800 GTX achieved the equivalent to between 50 M and 60 M splats per second. For comparison, Zhang and Pajarola [36] reported a performance of up to 24.9 M splats per second and Guennebaud et al. [35] reported 37.5 M splats per second—both for the same viewport size on the same GPU.

Our implementation rendered the line-based model of a single tree in Figure 13b at 31 frames per second, i.e., only slightly faster than the standard triangle rendering in Figure 13a, which achieves 28 frames per second. However, the pull-push interpolation of our approach is independent of the model size; thus, this part of the algorithm performs just as fast for the set of ten trees depicted in Figure 19b, which achieves a performance of 15 frames per second. The triangle rendering in Figure 19a, on the other hand, performs considerably slower at 9 frames per second.

7 Conclusion

While point-based and line-based graphics allow us to choose a dynamic level of detail for an efficient, adaptive sampling in object space, the reconstruction and shading of a scene should be performed in image space to achieve a better time complexity for large models. In this work, we have demonstrated an efficient GPU implementation of an image-space reconstruction for point-based and line-based graphics, which can be combined with polygonal graphics.

In general, our approach does not provide the performance of techniques based on textured polygons for very low detail representations, e.g., billboards, nor does it provide the image quality of high-resolution polygon models with (procedural) textures. However, it provides a good compromise between image quality and rendering performance for a large range of levels of detail. Moreover, it uses point-based and line-based models, which are particularly well suited for rendering with dynamic level of detail. Therefore, we consider our method to be a valuable complement to previously published rendering techniques and hope that it will help to address the challenges posed by increasing display resolutions in the future.

8 Acknowledgements

We would like to acknowledge the grant of the first author provided by Brazilian agency CNPq (National Counsel of Technological and Scientific Development).

References

[1] B. Watson, D. Luebke, The ultimate display: Where will all the pixels come from?, Computer 38 (8) (2005) 54–61.

[2] A. Dayal, C. Woolley, B. Watson, D. Luebke, Adaptive frameless rendering, in: EGSR '05: Proceedings of the Eurographics Symposium on Rendering, 2005, pp. 265–275.

[3] H. Pfister, M. Zwicker, J. van Baar, M. Gross, Surfels: Surface elements as rendering primitives, in: Proceedings SIGGRAPH '00, 2000, pp. 335–342.

[4] S. Rusinkiewicz, M. Levoy, QSplat: A multiresolution point rendering system for large meshes, in: Proceedings SIGGRAPH '00, 2000, pp. 343–352.

[5] B. Chen, M. X. Nguyen, Pop: A hybrid point and polygon rendering system for large data, in: VIS '01: Proceedings of IEEE Visualization '01, 2001, pp. 45–52.

[6] J. D. Cohen, D. G. Aliaga, W. Zhang, Hybrid simplification: Combining multi-resolution polygon and point rendering, in: VIS '01: Proceedings of IEEE Visualization '01, 2001, pp. 37–44.

[7] L. Coconu, H.-C. Hege, Hardware-accelerated point-based rendering of complex scenes, in: EGRW '02: Proceedings of the 13th Eurographics Workshop on Rendering, 2002, pp. 43–52.

[8] O. Deussen, C. Colditz, M. Stamminger, G. Drettakis, Interactive visualization of complex plant ecosystems, in: Proceedings of IEEE Visualization 2002, 2002, pp. 219–226.

[9] G. Guennebaud, L. Barthe, M. Paulin, Deferred splatting, Computer Graphics Forum 23 (3) (2004) 653–660.

[10] J. P. Grossman, W. J. Dally, Point sample rendering, in: 9th Eurographics Workshop on Rendering '98, 1998, pp. 181–192.

[11] L. Ren, H. Pfister, M. Zwicker, Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering, Computer Graphics Forum (Eurographics 2002) 21 (3) (2002) 461–470.

[12] M. Botsch, A. Wiratanaya, L. Kobbelt, Efficient high quality rendering of point sampled geometry, in: EGRW '02: Proceedings of the 13th Eurographics Workshop on Rendering, 2002, pp. 53–64.

[13] M. Botsch, L. Kobbelt, High-quality point-based rendering on modern GPUs, in: PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications, 2003, pp. 335–343.

[14] M. Zwicker, J. Räsänen, M. Botsch, C. Dachsbacher, M. Pauly, Perspective accurate splatting, in: GI '04: Proceedings of the 2004 Conference on Graphics Interface, 2004, pp. 247–254.

[15] M. Botsch, A. Hornung, M. Zwicker, L. Kobbelt, High-quality surface splatting on today's GPUs, in: Proceedings of the Eurographics/IEEE Symposium on Point-Based Graphics '05, 2005, pp. 17–24.

[16] J. Krüger, R. Westermann, Linear algebra operators for gpu implementation of numerical algorithms, ACM Transactions on Graphics 22 (3) (2003) 908–916.

[17] M. Strengert, M. Kraus, T. Ertl, Pyramid methods in GPU-based image processing, in: Proceedings of Vision, Modeling, and Visualization 2006, 2006, pp. 169–176.

[18] R. Marroquim, M. Kraus, P. R. Cavalcanti, Efficient point-based rendering using image reconstruction, in: Proceedings Symposium on Point-Based Graphics, 2007, pp. 101–108.

[19] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, P. Prusinkiewicz, Realistic modeling and rendering of plant ecosystems, in: SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, 1998, pp. 275–286.

[20] L. Kobbelt, M. Botsch, A survey of point-based techniques in computer graphics, Computers & Graphics 28 (6) (2004) 801–814.

[21] M. Sainz, R. Pajarola, Point-based rendering techniques, Computer & Graphics 28 (6) (2004) 869–879.

[22] M. Sainz, R. Pajarola, R. Lario, Points reloaded: Point-based rendering revisited, in: Proceedings of the Eurographics Symposium on Point-Based Graphics '04, 2004, pp. 121–128.

[23] M. Gross, H. Pfister (Eds.), Point-Based Graphics, Morgan Kaufmann Publishers, 2007.

[24] C. Csuri, R. Hackathorn, R. Parent, W. Carlson, M. Howard, Towards an interactive high visual complexity animation system, in: SIGGRAPH '79: Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques, 1979, pp. 289–299.

[25] A. R. Forrest, On the rendering of surfaces, in: SIGGRAPH '79: Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques, 1979, pp. 253–259.

[26] E. Catmull, J. Clark, Recursively generated B-spline surfaces on arbitrary topological meshes, Computer Aided Design 10 (6) (1978) 350–355.

[27] R. L. Cook, L. Carpenter, E. Catmull, The Reyes image rendering architecture, in: SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, 1987, pp. 95–102.

[28] M. Levoy, T. Whitted, The use of points as a display primitive, Technical Report TR 85-022, University of North Carolina at Chapel Hill (1985).

[29] P. J. Burt, Moment images, polynomial fit filters, and the problem of surface interpolation, in: Proceedings of Computer Vision and Pattern Recognition, 1988, pp. 144–152.

[30] S. J. Gortler, R. Grzeszczuk, R. Szeliski, M. F. Cohen, The Lumigraph, in: SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, 1996, pp. 43–54.

[31] V. Popescu, J. Eyles, A. Lastra, J. Steinhurst, N. England, L. Nyland, The WarpEngine: An architecture for the post-polygonal age, in: Proceedings SIGGRAPH '00, 2000, pp. 433–442.

[32] M. Zwicker, H. Pfister, J. van Baar, M. Gross, Surface splatting, in: Proceedings SIGGRAPH '01, 2001, pp. 371–378.

[33] G. Guennebaud, M. Paulin, Efficient screen space approach for hardware accelerated surfel rendering, in: Proceedings of Vision, Modeling and Visualization 2003, 2003, pp. 485–495.

[34] M. Botsch, M. Spernat, L. Kobbelt, Phong splatting, in: Proceedings of the Eurographics Symposium on Point-Based Graphics '04, 2004, pp. 25–32.

[35] G. Guennebaud, L. Barthe, M. Paulin, Splat/mesh blending, perspective rasterization and transparency for point-based rendering, in: Proceedings of the IEEE/Eurographics/ACM Symposium on Point-Based Graphics '06, 2006, pp. 49–58.

[36] Y. Zhang, R. Pajarola, Single-pass point rendering and transparent shading, in: Proceedings of the Eurographics/IEEE VGTC Symposium on Point-Based Graphics '06, 2006, pp. 37–48.

[37] Y. Zhang, R. Pajarola, Deferred blending: Image composition for single-pass point rendering, Computer & Graphics 31 (2) (2007) 175–189.

[38] T. Weyrich, S. Heinzle, T. Aila, D. Fasnacht, S. Oetiker, M. Botsch, C. Flaig, S. Mall, K. Rohrer, N. Felber, H. Kaeslin, M. Gross, A hardware architecture for surface splatting, ACM Transactions on Graphics (Proceedings ACM SIGGRAPH 2007) 26 (3).

[39] G. Schaufler, H. W. Jensen, Ray tracing point sampled geometry, in: Proceedings of the Eurographics Workshop on Rendering Techniques 2000, 2000, pp. 319–328.

[40] A. Adamson, M. Alexa, Ray tracing point set surfaces, in: SMI '03: Proceedings of Shape Modeling International 2003, 2003, pp. 272–279.

[41] I. Wald, H.-P. Seidel, Interactive ray tracing of point-based models, in: Proceedings of the Eurographics/IEEE VGTC Symposium on Point-Based Graphics '05, 2005, pp. 9–16.

[42] S. Lefebvre, S. Hornus, F. Neyret, Octree textures on the GPU, in: M. Pharr (Ed.), GPU Gems 2, Addison Wesley, 2005, pp. 595–613.

[43] R. Schnabel, S. Moser, R. Klein, A parallelly decodeable compression scheme for efficient point-cloud rendering, in: Proceedings Symposium on Point-Based Graphics, 2007, pp. 119–128.

[44] K.-H. Wong, X. Ouyang, C.-W. Lim, T.-S. Tan, J. Nievergelt, Rendering anti-aliased line segments, in: CGI '05: Proceedings of the Computer Graphics International 2005, 2005, pp. 198–205.

[45] M. Zöckler, D. Stalling, H.-C. Hege, Interactive visualization of 3d-vector fields using illuminated stream lines, in: VIS '96: Proceedings of IEEE Visualization '96, 1996, p. 107pp.

[46] O. Mallo, R. Peikert, C. Sigg, F. Sadlo, Illuminated lines revisited, in: Proceedings of IEEE Visualization '05, 2005, pp. 19–26.

[47] C. Stoll, S. Gumhold, H.-P. Seidel, Visualization with stylized line primitives, in: Proceedings of IEEE Visualization 2005, 2005, pp. 695–702.

[48] D. Merhof, M. Sonntag, F. Enders, C. Nimsky, G. Greiner, Hybrid visualization for white matter tracts using triangle strips and point sprites, IEEE Transactions on Visualization and Computer Graphics 12 (5) (2006) 1181–1188.

[49] J. T. Kajiya, T. L. Kay, Rendering fur with three dimensional textures, in: SIGGRAPH '89: Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques, 1989, pp. 271–280.

[50] D. C. Banks, Illumination in diverse codimensions, in: SIGGRAPH '94: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, 1994, pp. 327–334.

[51] M. Pharr, G. Humphreys, Physically Based Rendering: From Theory to Implementation, Morgan Kaufmann Publishers Inc., 2004.