Hardware-accelerated point-based rendering of surfaces and volumes

Eduardo Tejada Tobias Schafhitzel Thomas Ertl

Universität Stuttgart
{eduardo.tejada|tobias.schafhitzel|thomas.ertl}@vis.uni-stuttgart.de

ABSTRACT

In this paper, we present a fast GPU-based algorithm for ray-tracing point-based models, which includes an efficient computation of secondary and shadow rays, contrary to previous work which supported ray-surface intersections for primary rays only. Volumetric effects are added to the models by means of scattered data interpolation in order to combine point-based surface and volume rendering in the same scene. This allows us to obtain effects such as refraction within volumetric objects. The flexibility of our method is demonstrated by combining shadows, textured objects, refraction and volumetric effects in the same scene comprised uniquely by point sets.

Keywords: Point-based; Point Set Surfaces; Graphics hardware; Ray-tracing.

1 INTRODUCTION

Mesh-based methods have become the standard *de facto* for a wide range of applications. The advances in mesh-based techniques stimulated the development of hardware specific to meshes, which in turn encouraged the development of new and better mesh-based methods. However, during the last five years, the increase in the computation power of todays processors and the flexibility provided by graphics hardware, allowed the arising of a new set of point-based techniques, which represent an alternative to mesh-based methods. By working directly with point clouds, the processing is done on the raw data without the need for any intermediate representation and generally artificial connectivity relations[18]. Since the only information needed is the geometry given by the points, larger models can be stored in memory. Also, re-sampling during extreme deformations and topology changes is supported efficiently.

Among the point-based surface approximation methods developed in the last years are Point Set Surfaces (PSS)[6;30] As stated by Adamson and Alexa[2], PSS have clear advantages over other point-based surface approximation techniques when ray-tracing is to be used, namely the locality of the computations, the possibility of defining a minimum feature size and the fact that the surface is smooth and manifold. Beside the inherent implications of these three characteristics, the second advantage can be exploited when computing the intersection of the ray with the surface, whilst the last one makes CSG operations feasible.

However, ray-tracing a PSS on current standard PC processors is not interactive. To tackle this problem, implementations on special hardware have been presented. Wald and Seidel[28] implemented a highly-tuned ray-tracer that used Adamson and Alexa's implicit surface definition[3] on an Opteron PC, making use of SIMD operations and a number of acceleration techniques originally developed by the authors in the last years for mesh models. Tejada et al.[27] presented a GPU implementation to compute ray-surface intersections of primary with the PSS proposed by Alexa et al.[6].

Thus, based on the work by Tejada et al., we chose to render the PSS on the GPU exploiting the flexibility and new functionality offered by currently available commodity graphics hardware. As mentioned above, this approach includes a fast computation of the intersection of primary rays with the PSS. However, intersections with secondary and shadow rays are computed by means of a brute force scheme. Therefore, computing efficiently these intersections with the PSS on the GPU is our main concern in this work. The resulting algorithm, which we present in this paper, allows us to work with nested surfaces and to include self-shadowing in the scene. In order to include volumetric effects in the scene, we exploited the texture-based data structure we use for ray-tracing the PSS, to interpolate the scalar at the sampling points. This way, we are able to create scenes, where both surface rendering of the PSS and volumetric effects are combined. Furthermore, if the points belonging to a material boundary of the volumetric object are properly identified, we are able to combine refraction with volume rendering to obtain interesting visual effects.

2 RELATED WORK

As mentioned before, during the last years point-based methods have gained a new popularity. Modeling tools and techniques for creating and editing point-based models have been developed[12;20;30]. Also deformable bodies simulation[16], animation[19;21], and rendering algorithms[1;3;9;25;29] have been proposed. Alexa et al.[6] proposed a resampling technique for generating dense samplings in order to cover the image space consistently by simply projecting the points onto the screen. The 'surfaces' obtained this way are therefore known as Point Set Surfaces (PSS). PSS is a surface approximation method that, as stated in the previous section, presents important advantages for ray-tracing. PSS have been further studied by Alexa et al.[5], by Adamson and Alexa[3;4], and by Amenta and Kil[7;8], who noted that PSS are an special case of *extremal surfaces*[15]. Amenta and Kil[7] also studied important characteristics about the domain of point set surfaces.

In general, PSS methods make use of classical differential geometry results to ensure consistent local representation through polynomial approximations. In this respect, Zwicker et al.[30] use linear minimization functions based on Weighted Least-Squares (WLS) to define the local polynomial functions, whilst Alexa et al.[6] employ a non-linear strategy based on Moving Least-Squares (MLS) to define a local coordinate system on which a local polynomial approximation of the surface is calculated using WLS. This approach was based on the work by Levin[14] on the approximation power of the MLS method. Point-based techniques that make use of implicit functions whose zero set is guaranteed to generate surface representations were also proposed[3;8;13]. Also, guarantees for homeomorphical approximations to the original object based on this implicit function have been presented[11].

Ray-tracing PSS has been previously addressed both on the CPU[2;28] and the GPU[27]. However, as already mentioned, ray-tracing a PSS on the CPU is prohibitively slow. Therefore, Wald and Seidel[28] presented a ray-caster, implemented on the Opteron PC, for the implicit surface definition by Adamson and Alexa[3]. Also, a GPU-based technique was proposed[27], that efficiently computes the intersection of primary rays with the surface. Although, the advantage of this implementation is the use of commodity hardware, a description of how to compute the intersection between secondary and shadow rays with the PSS efficiently is not presented. Ray-tracing on the GPU has been addressed in a more general sense by Purcell[23].

3 POINT SET SURFACES

Given a set of points $p_i \in \mathbb{R}^3, i \in \{1,...,N\}$, Alexa et al.[6] define the corresponding Point Set Surface S_p as the set of stationary points for the projection procedure described in the following. First a local orthonormal coordinate system is built upon the plane $H(n, r+tn)$, where the normal n to the plane and the scalar t are obtained by minimizing

$$\sum_{p_i \in P(r)} < p_i - r - tn, n >^2 \theta(\|p_i - r - tn\|) \qquad (1)$$

with respect to both n and t. Here $\theta(x)$ is a non-negative, monotonically decreasing function and $P(r)$ is the set of points in the neighborhood of r. For the weighting function $\theta(x)$, authors frequently make use of a Gaussian $\theta(x) = e^{-\frac{x^2}{h^2}}$, where h represents the local level of detail of the object (feature size). Note that the constraint $\|n\| = 1$ must be observed during the minimization. Also note that, as a result of this minimization, H will be a plane close to r and quasi-tangent to S_p.

Once the local coordinate system is built over H with origin in $q = r + tn$, a local bivariate polynomial approximation g to the surface is computed using the points in $P(r)$. If q_i is the projection of p_i onto H, (x_i, y_i) is the local representation of q_i in the local system and $f_i = n \cdot (p_i - q)$ (the height of p_i over H), then the coefficients of this polynomial are found by minimizing

$$\sum_{p_i \in P} (g(x_i, y_i) - f_i)^2 \theta(\|p_i - q\|). \qquad (2)$$

Then, the projection $\mathscr{P}(r)$ onto S_p is defined by $\mathscr{P}(r) = q + g(0,0)$.

The minimization defined in Equation 1 is a non-linear optimization problem. To solve it, we use an iterative process that descends to the next local minima. This iterative process consists of two steps: (1) minimize with respect to t and (2) minimize with respect to n. In order to start the process a first approximation of n must be computed using covariance analysis [18]. Then, the minimization with respect to t is performed using the *Brent with derivative* algorithm[22], and the resulting t is then fixed for minimizing with respect to n. Since minimizing with respect to n in $\mathbb{S}^2$ (the space of directions) is computational expensive, an approximation could be obtained as in the work by Alexa et al.[6] by minimizing q on the plane defined by $q = \mathscr{P}(r) + tn$ and the previous n using the *conjugate gradient* algorithm[26]. These two steps are repeated until the change in both parameters is smaller than a pre-defined threshold. With these results we build the local coordinate system and compute the local polynomial approximation as described above. The minimization of Equation 2 can also be performed using *conjugate gradient*.

4 RAY-TRACING POINT SET SURFACES ON THE GPU

Based on the observation that the influence of a sample point p_i on the shape of the PSS can be limited, Adamson and Alexa[2] defined a trust region T_i for each sample point p_i as the ball with center on p_i and radius b, where $0.5h < b < h$. The idea for the ray-tracing process is to compute a first approximation of the intersection between the ray and the PSS and then converge iteratively to the actual intersection. For this, in a pre-processing step, the local polynomial approximation at each point p_i is calculated as described above, i.e. $r = p_i$. The first approximation for the intersection is then the nearest intersection x_t of the ray with these pre-computed local polynomials, that lies within the trust region of the corresponding p_i. To converge to the actual intersection between the ray and S_p, $\mathscr{P}(x_t)$ is computed. Note that during this projection process a local polynomial approximation g_t is found. The distance between $\mathscr{P}(x_t)$ and x_t gives us a measure for the error e_t of the current intersection x_t. If this error is greater than an user-defined threshold, the intersection between g_t and the ray is calculated and taken as the new x_t. If x_t lies outside the current trust region, the next nearest intersection of the ray with the pre-stored local polynomials is found and the process repeated.

Some simplifications were introduced into the GPU implementation of the projection operator[27]. Firstly, since the pre-computation of the local polynomials stored in each p_i requires projecting the sample points, which are almost on S_p, the iterative process for minimizing Equation 1 is not performed. Instead, the initial n is calculated using covariance analysis and the local coordinate system is built over $H(n, p_i)$. The same applies to the computation of $\mathscr{P}(x_t)$ for the current intersection x_t. Secondly, in order to make feasible the computation of the coefficients of the local polynomial approximations on the GPU, an incomplete polynomial of degree 2 is used, namely $g(x,y) = Ax^2 + By^2 + Cxy + D$.

We introduce here a further simplification based on the observation that the intersection of a ray with the stored polynomials often suffices for obtaining good approximations to the actual intersection point between the ray and the point set surface. This means, that the iterative process used to converge to the actual intersection with S_p is completely avoided. As can be appreciated in Section 7, the resulting renderings are of good quality even with this first rough approximation, whilst gaining performance.

To store the information needed on the GPU, we use a 2D texture `tex_positions` with the positions in space of the sampling points p_i and a 3D texture `tex_neighbors` with the pointers to the nearest neighbors of each p_i. These textures are used to calculate the coefficients of the polynomial $g_i(x,y)$ for each p_i, which is written to the target texture `tex_polynomial`, and the orthonormal base of the local coordinate system, written to textures `tex_basea`, `tex_baseb` and `tex_basec`, as done in previous work[27].

With this information, the intersection of the primary rays with S_p can be efficiently computed[27]. However, contrary to previous work, we approximate this intersection as the intersection of the ray with the nearest polynomial that lies inside the corresponding trust region. The process is started by rendering for each p_i a viewport-aligned disc with center in p_i and radius b. Each fragment generated this way represents a ray that intersects the trust region T_i of p_i. In the fragment shader, the ray corresponding to the fragment is transformed to the local system, its intersection with the local polynomial g_i is computed and, in case the intersection lies within T_i, the intersection point is written to texture `tex_intersection`, and the normal at the intersection point to texture `tex_normal`. With this information the viewable part of the surface can be lit and displayed, as shown in Figure 1 where the surface rendering of the Armadillo model (172974 points) obtained with our GPU renderer is shown. The performance for this model was 6 fps on an Nvidia 7900 GT grahics card.

5 SHADOWS, REFLECTIONS AND REFRACTION ON THE GPU

To limit the number of points tested to find the intersection of secondary rays with the local polynomials, we build a Cartesian grid of reduced resolution (about 20^3 for a point set of size 35000) covering the domain of the point set as depicted in Figure 3a. Then, to find the intersection of a given ray with the point set surface we traverse the grid starting in the cell where the origin of the ray lies and test the points stored in each traversed cell. If the origin of the ray lies outside the grid, the intersection of the ray with the bounding box is found and used as the new origin of the ray.

One important issue to consider is shown in Figure 3b, where the case of a polynomial of a point in a neighboring cell intersecting the current cell is exemplified. To deal with this case during the test we must include in each cell the points in neighboring cells that lie within a distance b from the boundaries of the cell as shown in Figure 3a.

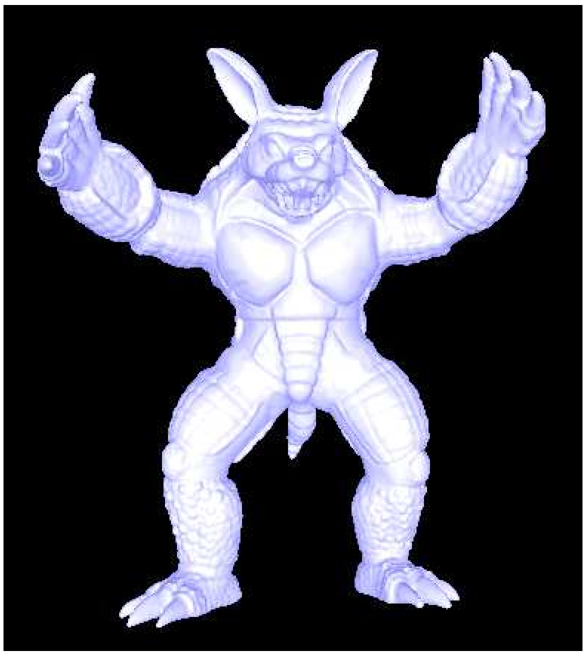


Figure 1: The Armadillo model rendered with our GPU-based point set surfaces renderer.

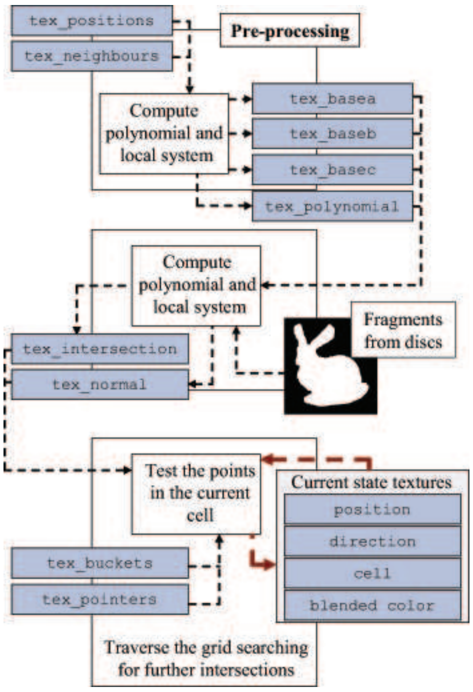


Figure 2: Ray-tracing a PSS following the secondary refracted rays only. The red pointed arrows represent the loop on the CPU used for reducing the number of instructions performed in a single pass.

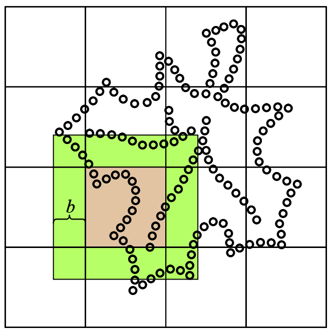
This process is also performed on the GPU. Thus, a 3D texture, `tex_buckets`, for the grid is created where a pointer to a position in an intermediary 2D texture, `tex_pointers`, is stored. In texture `tex_pointers` pointers to the positions in `tex_position` and `tex_neighbors` corresponding to the points in the cell are stored sequentially. Then, given a fragment corresponding to an arbitrary ray (generated by rendering a single quad covering the viewport and following the refracting ray) the points corresponding to the cells intersected by the ray are tested in the fragment shader using this texture-based data structure.

In Figure 2 we show how this can be used for ray-tracing a PSS following only the secondary refracted rays. A complete ray-tracing algorithm on the GPU could be easily implemented from this results (See [10;24] for details on the implementation of a GPU-based ray-tracer). One important issue shown in Figure 2 is the need for a CPU loop for traversing the grid. This loop could also be implemented in a fragment shader and only one render pass would be necessary for traversing the complete object finding all intersections of the ray (refracted each time an intersection is found) with the object. However, the number of instructions executed by the fragment processor is limited and when the number of points in the object over-passes a certain limit, regions with green pixels are obtained. Therefore, we opted for performing a render pass for each traversed cell. We use ping-pong rendering in order to fetch the results of the previous pass in the current pass, which included the position, direction, cell and accumulated color (therefore, we render to four render targets).

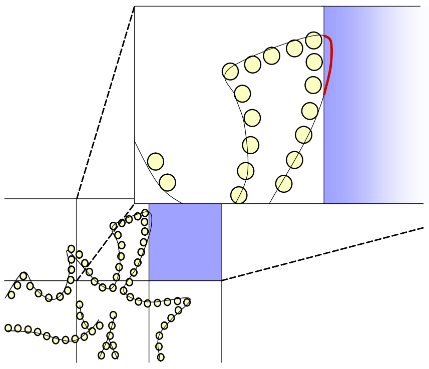
Also, support for different indices of refraction for each channel was implemented. Since it is only possible to have four render targets, we have to execute the process above described for each channel and combine their results in a further render pass. Shadows, including self-shadowing, are easily implemented by traversing the grid following the light vector. Figure 4 shows visual results of self-shadowing and shadows between the Stanford Bunny and the point set of an artifical terrain.

6 INTRODUCING VOLUME RENDERING EFFECTS

As mentioned before, we also included volumetric effects in the scene. For this, we make use of the texture-based data structure to perform scattered data interpolation[17] of scalar values stored at the sampling points. These points are classified as belonging to a material boundary or not. The integration for a ray is started by using the boundary points to find the first intersection x_p of the ray with the object, with the process described in the previous section. The ray is then sampled using a constant step. The neighbors of the current sampling point x_p are found using the grid described in the previous section, by accessing the points in the cell where x_p is stored, as well as the points belonging to



(a)



(b)

Figure 3: The Cartesian grid to traverse the object. In (a) the points within the bright green and red areas are included in the list of points of the cell colored light red, due to the case shown in (b) for the shaded cell. Although there is no point inside the cell, a segment (shown in red) of the Point Set Surface could intersect it.

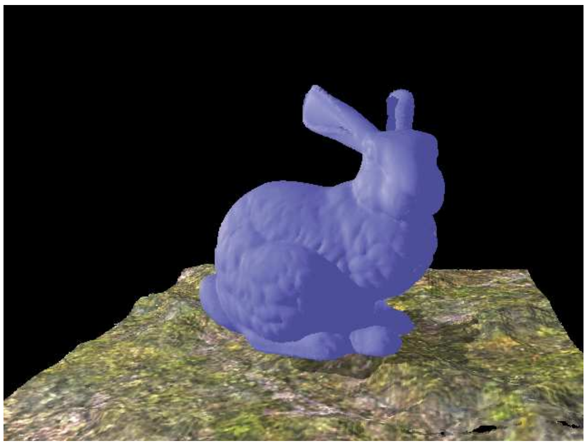


Figure 4: The Stanford Bunny model generating a shadow on the point-based texturized terrain. Self-shadowing is also computed.

the neighboring cells. From these points we only use those within a pre-defined distance k to the integration point. The interpolated value $s(x_p)$ at x_p is then given by

$$s(x_p) = \frac{\sum_{p_i \in N(x_p)} s(p_i) \exp(-d(x_p,p_i)^2/2\rho)}{\sum_{p_i \in N(x_p)} \exp(-d(x_p,p_i)^2/2\rho)} \qquad (3)$$

where $N(x_p)$ is the set of points in the k-neighborhood of x_p, $d(x_p,p_i)$ is the distance between x_p and p_i and ρ is a smoothing parameter.

This process is easily combined with the one depicted in Figure 2 to render mixed scenes with both point-based surfaces and scattered volumetric data. Furthermore, refraction at the boundaries of a volume is possible. During grid traversal, for each visited cell, we test the intersection between the ray and the polynomials of the points stored in the cell that belong to a material boundary. If an intersection is found, the direction of the refracted ray is computed and the integration process continues with this new ray direction. This rendering process results in interesting visual effects, as can be seen in the next section.

7 RESULTS

Here we present performance and visual results obtained with known data sets from tests carried out on a PC equipped with an NVidia 7900 GT graphics card and rendering to a viewport of size 640×480. We run tests with the models using refraction, shadows computation, volume rendering and volume rendering with refraction. Also, results for surface rendering with different indices of refraction (IOR) for each channel are presented. All these rendering modes can be seen in Figure 7. For surface rendering without refraction we achieved frames rates of 28 and 20 fps for the Stanford Bunny dataset (35947 points) and the Horse dataset (48485 points) respectively. Including shadows reduced the performance to 10 fps for the Bunny and to 11 fps for the Horse.

Complex scenes can be described by means of point clouds as seen in the figures, where examples with texturized point-based terrains, scenes with refracted surfaces and volumes, self-shadowing and refracted volumes are shown. Furthermore, scenes with combined modalities, e.g. refracted volumes and texturized terrains, are supported. Following the refracted ray through the object we obtained frame rates of 1.33 fps for the Stanford Bunny and of 2.16 fps for the Horse model. As stated above, multiple indices of refraction were also implemented using multiple passes, which turned into a slow down of the rendering process (0.25 and 0.22 fps for the Bunny and the Horse respectively).

On the other hand, including volumetric effects in the rendering does not have a significant impact on the performance, since the scalar values are interpolated at the

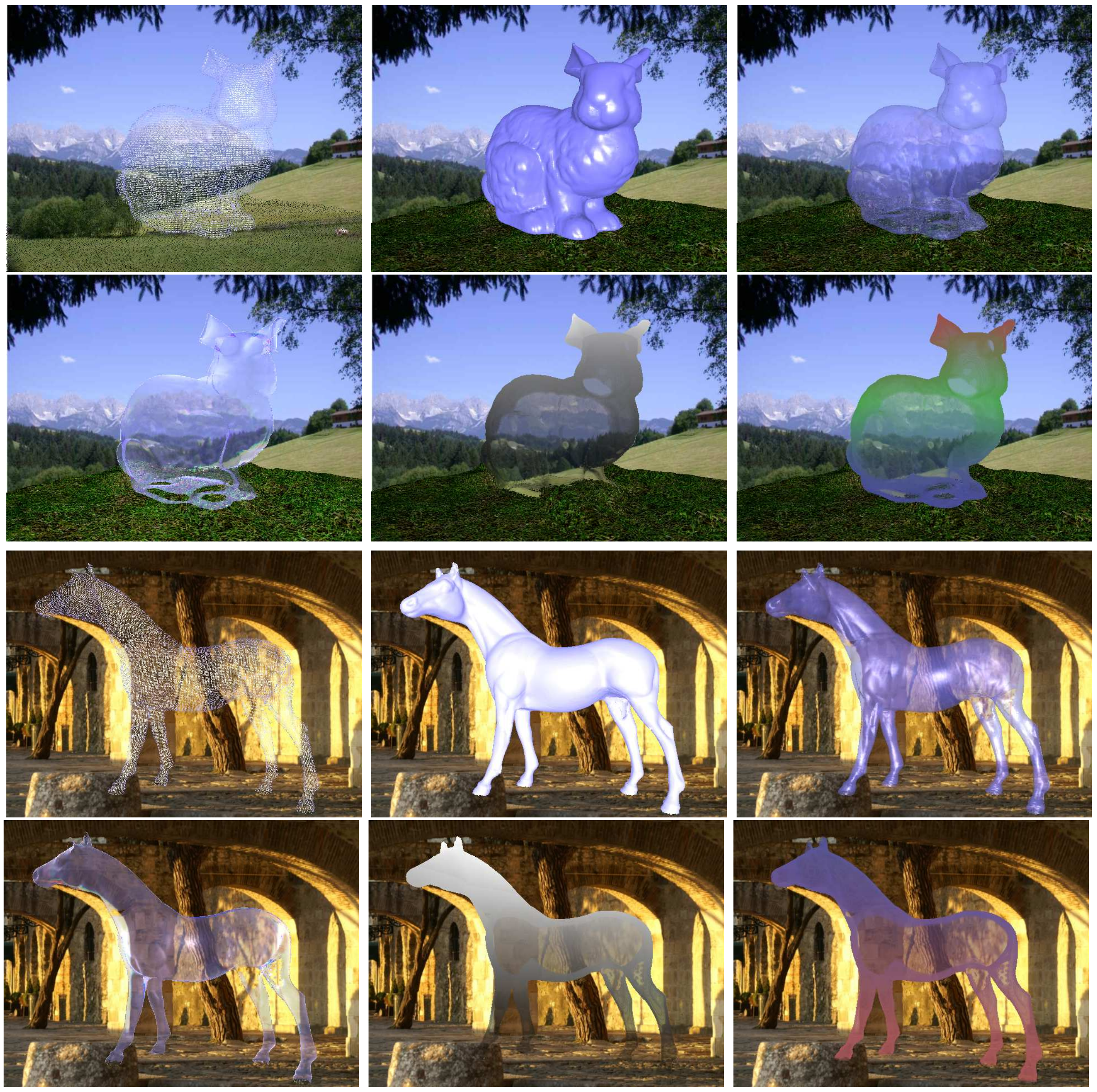


Figure 5: The Stanford Bunny and Horse point sets rendered with the techniques implemented. Starting with the left-top image row-wise: the raw point set, the point set surface, refraction with single IOR, refraction with multiple IOR, volume rendering with refraction and volume rendering using transfer functions.

same positions on the ray we use to traverse the grid structure while computing the intersections of the ray with the surface. Thus, 1.08 fps were achieved for the Bunny using combined refraction and volumetric effects. In this case, the performance for the Horse was 0.96 fps.

8 CONCLUSION

In this paper, we presented GPU-based rendering techniques to render scenes comprised by volumes and surfaces described only by points with no other information than the position of the points in space and the scalar value associated to them. By implementing our techniques on the GPU, we managed to reduce the computation time compared to CPU implementations. Although higher frame rates have been achieved with implementations on specialized hardware, the advantage of working with commodity graphics hardware is the accessibility to desktop users and the rapid increase in computational power and flexibility compared to CPUs.

Currently the most important problem we find for continuing with the development of point-based methods is the complexity of the proximity queries. In order to reduce the time spent in this queries, we had to precompute the nearest neighbors for the sample points and use a grid-based structure during ray traversal as in previous work on ray-tracing on the GPU. However, this has the disadvantage of using additional storage space that, although is not as large as the required by surface meshes or structured and non-structured volumes, could become a serious problem for larger models. On the other hand, we believe that the advances on point-based methods will capture the attention of researchers and hardware manufacturers in the near future, resulting in the inclusion of such expensive operations in hardware allowing the further developing of point-based computer graphics.

ACKNOWLEDGMENTS

Research supported in part by the DAAD (German Academic Exchange Service) grant Nr. A/04/08711.

REFERENCES

[1] Bart Adams, Richard Keiser, Mark Pauly, Leonidas J. Guibas, Markus Gross, and Philip Dutré. Efficient raytracing of deforming point-sampled surfaces. *Computer Graphics Forum*, 24(3):677–684, 2005.

[2] A. Adamson and M. Alexa. Ray tracing point set surfaces. In *Proc. of Shape Modeling International*, page 298. IEEE Computer Society, 2003.

[3] Anders Adamson and Marc Alexa. Approximating and intersecting surfaces from points. In *Proc. of Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 230–239. Eurographics Assoc., 2003.

[4] M. Alexa and A. Adamson. On normals and projection operators for surfaces defined by point sets. In *Eurographics Symposium on Point-Based Graphics*, pages 149–155. Eurographics Assoc., 2004.

[5] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. Silva. Computing and rendering point set surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):3–15, 2003.

[6] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. In *Proc. of IEEE Visualization*, pages 21–28. IEEE Computer Society, October 2001.

[7] Nina Amenta and Yong J Kil. The domain of a point set surfaces. *Eurographics Symposium on Point-based Graphics*, 1(1):139–147, 2004.

[8] Nina Amenta and Yong Joo Kil. Defining point-set surfaces. *ACM Transactions on Graphics*, 23(3):264–270, 2004.

[9] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern GPUs. In *Proc. of Pacific Conference on Computer Graphics and Applications*, page 335. IEEE Computer Society, 2003.

[10] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[11] Tamal K Dey, S Goswami, and J Sun. Extremal surface based projections converge and reconstruct with isotopy. Technical Report OSU-CISRC-05-TR25, Ohio State University, 2005.

[12] Leif Kobblet and Mario Botsch. A survey of point-based techniques in computer graphics. *Computer & Graphics*, 28(6):801–814, 2004.

[13] Ravikrishna Kolluri. Provably good moving least squares. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1008–1017, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics. Vencedor do prêmio: *Best Student Paper Award*.

[14] David Levin. The approximation power of moving least-squares. *Mathematics of Computation*, 67(224):1517–1531, 1998.

[15] Gerard Medioni, Mi-Suen Lee, and Chi-Keung Tang. *Computational Framework for Segmentation and Grouping*. Elsevier Science, 2000.

[16] M. Müller, R. Keiser, A. Nealen, M. Pauly, M. Gross, and M. Alexa. Point based animation of elastic, plastic and melting objects. In *Proc. of ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 141–151. ACM Press, 2004.

[17] Gregory M. Nielson. Scattered data modeling. *IEEE Computer Graphics and Applications*, 13(1):60–70, 1993.

[18] Mark Pauly. *Point Primitives for Interactive Modeling and Processing of 3D Geometry*. PhD thesis, Federal Institute of Technology (ETH) of Zurick, 2003.

[19] Mark Pauly, Richard Keiser, Bart Adams, Philip Dutré, Markus Gross, and Leonidas J. Guibas. Meshless animation of fracturing solids. *ACM Transactions on Graphics*, 24(3):957–964, 2005.

[20] Mark Pauly, Richard Keiser, Leif P. Kobbelt, and Markus Gross. Shape modeling with point-

sampled geometry. *ACM Transactions on Graphics*, 22(3):641–650, 2003.

[21] Mark Pauly, Dinesh K. Pai, and Leonidas J. Guibas. Quasi-rigid objects in contact. In *Proc. of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 109–119. ACM Press, 2004.

[22] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1st edition, 1986.

[23] T. Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004.

[24] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[25] Gernot Schaufler and Henrik Wann Jensen. Ray tracing point sampled geometry. In *Proc. of Eurographics Workshop on Rendering Techniques*, pages 319–328. Springer-Verlag, 2000.

[26] Jonathan R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, 1994.

[27] E. Tejada, J.P. Gois, L. G. Nonato, A. Castelo, and T. Ertl. Hardware-accelerated Extraction and Rendering of Point Set Surfaces. In *Proceedings of EUROGRAPHICS - IEEE VGTC Symposium on Visualization*, pages 21–28, 2006.

[28] Ingo Wald and Hans-Peter Seidel. Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics*, 2005.

[29] Michael Wand and Wolfgang Straßer. Multiresolution point-sample raytracing. In *Graphics Interface*, pages 139–148, 2003.

[30] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus Gross. Pointshop 3D: an interactive system for point-based surface editing. In *SIGGRAPH : Proc. of Computer Graphics and Interactive Techniques*, pages 322–329. ACM Press, 2002.