ISSN 1213-6972 *Volume 20, Number 3, 2012*

Journal of WSCG

An international journal of algorithms, data structures and techniques for computer graphics and visualization, surface meshing and modeling, global illumination, computer vision, image processing and pattern recognition, computational geometry, visual human interaction and virtual reality, animation, multimedia systems and applications in parallel, distributed and mobile environment.

Vaclav Skala – Union Agency

Journal of WSCG

Editor-in-Chief: Vaclav Skala
c/o University of West Bohemia
Faculty of Applied Sciences
Univerzitni 8
CZ 306 14 Plzen
Czech Republic
http://www.VaclavSkala.eu

Managing Editor: Vaclav Skala

Printed and Published by:
Vaclav Skala - Union Agency
Na Mazinach 9
CZ 322 00 Plzen
Czech Republic

Hardcopy: ***ISSN 1213 – 6972***
CD ROM: ***ISSN 1213 – 6980***
On-line: ***ISSN 1213 – 6964***

Journal of WSCG

Editor-in-Chief

Vaclav Skala

c/o University of West Bohemia
Centre for Computer Graphics and Visualization
Univerzitni 8
CZ 306 14 Plzen
Czech Republic

http://www.VaclavSkala.eu

Journal of WSCG URLs: http://www.wscg.eu or http://wscg.zcu.cz/jwscg

Editorial Advisory Board
MEMBERS

Journal of WSCG 2012

Board of Reviewers

Abad,F. (Spain)
Adzhiev,V. (United Kingdom)
Ariu,D. (Italy)
Assarsson,U. (Sweden)
Aveneau,L. (France)
Barthe,L. (France)
Battiato,S. (Italy)
Benes,B. (United States)
Benger,W. (United States)
Bengtsson,E. (Sweden)
Benoit,C. (France)
Beyer,J. (Saudi Arabia)
Biasotti,S. (Italy)
Bilbao,J. (Spain)
Biri,V. (France)
Bittner,J. (Czech Republic)
Bosch,C. (Spain)
Bouatouch,K. (France)
Bourdin,J. (France)
Bourke,P. (Australia)
Bruckner,S. (Austria)
Bruder,G. (Germany)
Bruni,V. (Italy)
Buriol,T. (Brazil)
Cakmak,H. (Germany)
Capek,M. (Czech Republic)
Cline,D. (United States)
Coquillart,S. (France)
Corcoran,A. (Ireland)
Cosker,D. (United Kingdom)
Daniel,M. (France)
Daniels,K. (United States)
de Geus,K. (Brazil)
De Paolis,L. (Italy)
Debelov,V. (Russia)
Dingliana,J. (Ireland)
Dokken,T. (Norway)
Drechsler,K. (Germany)
Durikovic,R. (Slovakia)
Eisemann,M. (Germany)
Erbacher,R. (United States)
Erleben,K. (Denmark)
Essert,C. (France)
Faudot,D. (France)
Feito,F. (Spain)
Ferguson,S. (United Kingdom)
Fernandes,A. (Portugal)
Flaquer,J. (Spain)
Flerackers,E. (Belgium)
Fuenfzig,C. (Germany)
Galo,M. (Brazil)
Garcia Hernandez,R. (Spain)
Garcia-Alonso,A. (Spain)
Gavrilova,M. (Canada)
Giannini,F. (Italy)
Gobron,S. (Switzerland)
Gonzalez,P. (Spain)
Gudukbay,U. (Turkey)
Guérin,E. (France)
Hall,P. (United Kingdom)
Hansford,D. (United States)
Haro,A. (United States)
Hasler,N. (Germany)
Hast,A. (Sweden)
Havran,V. (Czech Republic)
Hege,H. (Germany)
Hernandez,B. (Mexico)
Herout,A. (Czech Republic)
Hicks,Y. (United Kingdom)
Horain,P. (France)
House,D. (United States)
Chaine,R. (France)
Chaudhuri,D. (India)
Chmielewski,L. (Poland)
Choi,S. (Korea)
Chover,M. (Spain)
Chrysanthou,Y. (Cyprus)
Ihrke,I. (Germany)
Jansen,F. (Netherlands)
Jeschke,S. (Austria)
Jones,M. (United Kingdom)
Juettler,B. (Austria)
Kanai,T. (Japan)
Kim,H. (Korea)
Klosowski,J. (United States)
Kohout,J. (Czech Republic)
Krivanek,J. (Czech Republic)
Kurillo,G. (United States)
Kurt,M. (Turkey)
Lay Herrera,T. (Germany)
Lien,J. (United States)
Liu,S. (China)
Liu,D. (Taiwan)
Loscos,C. (France)
Lucas,L. (France)
Lutteroth,C. (New Zealand)
Maciel,A. (Brazil)
Madeiras Pereira,J. (Portugal)
Magnor,M. (Germany)
Manak,M. (Czech Republic)
Manzke,M. (Ireland)
Mas,A. (Spain)
Masia,B. (Spain)
Masood,S. (United States)
Matey,L. (Spain)
Matkovic,K. (Austria)
Max,N. (United States)
McDonnell,R. (Ireland)
McKisic,K. (United States)
Mestre,D. (France)
Molina Masso,J. (Spain)
Molla Vaya,R. (Spain)
Montrucchio,B. (Italy)
Muller,H. (Germany)

Murtagh,F. (Ireland)
Myszkowski,K. (Germany)
Niemann,H. (Germany)
Okabe,M. (Japan)
Oliveira Junior,P. (Brazil)
Oyarzun Laura,C. (Germany)
Pala,P. (Italy)
Pan,R. (China)
Papaioannou,G. (Greece)
Paquette,E. (Canada)
Pasko,A. (United Kingdom)
Pasko,G. (United Kingdom)
Pastor,L. (Spain)
Patane,G. (Italy)
Patow,G. (Spain)
Pedrini,H. (Brazil)
Peters,J. (United States)
Peytavie,A. (France)
Pina,J. (Spain)
Platis,N. (Greece)
Plemenos,D. (France)
Poulin,P. (Canada)
Puig,A. (Spain)
Reisner-Kollmann,I. (Austria)
Renaud,c. (France)
Reshetov,A. (United States)
Richardson,J. (United States)
Rojas-Sola,J. (Spain)
Rokita,P. (Poland)
Rudomin,I. (Mexico)
Runde,C. (Germany)
Sacco,M. (Italy)
Sadlo,F. (Germany)
Sakas,G. (Germany)
Salvetti,O. (Italy)
Sanna,A. (Italy)
Santos,L. (Portugal)
Sapidis,N. (Greece)
Savchenko,V. (Japan)
Sellent,A. (Germany)
Sheng,B. (China)
Sherstyuk,A. (United States)
Shesh,A. (United States)
Schultz,T. (Germany)
Sirakov,N. (United States)
Skala,V. (Czech Republic)
Slavik,P. (Czech Republic)
Sochor,J. (Czech Republic)
Solis,A. (Mexico)
Sourin,A. (Singapore)
Sousa,A. (Portugal)
Sramek,M. (Austria)
Staadt,O. ()
Stroud,I. (Switzerland)
Subsol,G. (France)
Sunar,M. (Malaysia)
Sundstedt,V. (Sweden)
Svoboda,T. (Czech Republic)
Szecsi,L. (Hungary)
Takala,T. (Finland)
Tang,M. (China)
Tavares,J. (Portugal)
Teschner,M. (Germany)
Theussl,T. (Saudi Arabia)
Tian,F. (United Kingdom)
Tokuta,A. (United States)
Torrens,F. (Spain)
Triantafyllidis,G. (Greece)
TYTKOWSKI,K. (Poland)
Umlauf,G. (Germany)
Vavilin,A. (Korea)
Vazquez,P. (Spain)
Vergeest,J. (Netherlands)
Vitulano,D. (Italy)
Vosinakis,S. (Greece)
Walczak,K. (Poland)
WAN,L. (China)
Wang,C. (Hong Kong SAR)
Weber,A. (Germany)
Weiss,G. (Germany)
Wu,E. (China)
Wuensche,B. (New Zealand)
Wuethrich,C. (Germany)
Xin,S. (Singapore)
Xu,D. (United States)
Yang,X. (China)
Yoshizawa,S. (Japan)
YU,Q. (United Kingdom)
Yue,Y. (Japan)
Zara,J. (Czech Republic)
Zemcik,P. (Czech Republic)
Zhang,X. (Korea)
Zhang,X. (China)
Zillich,M. (Austria)
Zitova,B. (Czech Republic)
Zwettler,G. (Austria)

Journal of WSCG

Vol.20, No.3

Contents

On-the-fly Luminance Correction for Rendering of Inconsistently Lit Point Clouds

Thomas Kanzok
Chemnitz University of Technology
Department of Computer Science
Visual Computing Laboratory
Straße der Nationen 62
09111 Chemnitz, Germany
thomas.kanzok@informatik.tu-chemnitz.de

Lars Linsen
Jacobs University
School of Engineering & Science
Visualization and Computer Graphics Laboratory
Campus Ring 1
28759 Bremen, Germany
l.linsen@jacobs-university.de

Paul Rosenthal
Chemnitz University of Technology
Department of Computer Science
Visual Computing Laboratory
Straße der Nationen 62
09111 Chemnitz, Germany
paul.rosenthal@informatik.tu-chemnitz.de

ABSTRACT

Scanning 3D objects has become a valuable asset to many applications. For larger objects such as buildings or bridges, a scanner is positioned at several locations and the scans are merged to one representation. Nowadays, such scanners provide, beside geometry, also color information. The different lighting conditions present when taking the various scans lead to severe luminance artifacts, where scans come together. We present an approach to remove such luminance inconsistencies during rendering. Our approach is based on image-space operations for both luminance correction and point-cloud rendering. It produces smooth-looking surface renderings at interactive rates without any preprocessing steps. The quality of our results is similar to the results obtained with an object-space luminance correction. In contrast to such an object-space technique the presented image-space approach allows for instantaneous rendering of scans, e.g. for immediate on-site checks of scanning quality.

Keywords

Point-cloud Rendering, Image-space Methods, Luminance Correction, Color-space Registration

1. INTRODUCTION

In the field of civil engineering large structures like bridges have to be surveyed on a regular basis to document the present state and to deduct safety recommendations for repairs or closures. Typically this is done by measuring the structures manually or semiautomatically at predefined measuring points and adding detailed photographs or by using completely automatic 3D scanning techniques.

Nowadays, both dominant surface digitalization techniques, laser scanning [BR02] as well as photogrammetry [SSS06,TS08], produce colored point clouds where the color of each point matches the color of the respective surface part under the lighting conditions at the time of scanning. Many photographs become redundant with this additional information. However, when dealing with large structures one has to do several scans from different points of view in order to generate a complete building model of desired resolution. As in most cases only one 3D scanner is used and relocated for each scan, the time of day and therefore the lighting conditions may differ significantly between adjacent scans or, on a cloudy day, even within one scan.

When combining the different scans to one object representation and using a standard geometry-based registration approach, the resulting point cloud may have severe inconsistencies in the luminance values. Because of the scanning inaccuracies in the geometric measures, the renderings of registered scans exhibit disturbing patterns of almost randomly changing luminance assignment in regions where scans with different lighting conditions overlap. We present an approach to correct the luminance and create a consistent rendering. "Consistent" in this context

means that no local luminance artifacts occur; it does not mean that the global illumination in the rendering is consistent.

A simple, yet effective approach to adjust the luminance is to average the luminances locally within neighborhoods in object space, as outlined in Section 4. We use this approach as a reference to the image-space approach we propose in Section 5. The reason for introducing the image-space approach is that the luminance correction in object space is rather time-consuming and needs to be done in a preprocessing step. The engineers, however, would like to get an immediate feedback during their field trip whether the scans they have taken are capturing all important details and are of sufficient quality. Hence, an immediate rendering of the combined scans is required. Our image-space approach allows for such an instantaneous investigation of the scans since it is capable of producing high-quality renderings of inconsistently lit point clouds at interactive framerates.

2. RELATED WORK

Today both the amount and size of generated surface data are steadily increasing. Beginning with the Digital Michelangelo project [LPC+00], which was the first one generating massive point clouds, the scanning hardware was getting significantly cheaper while producing results of increasing resolution and quality. The datasets that are typically generated these days range from hundreds of million to several billion surface points [WBB+08].

In this setting it is obvious, that global reconstruction of the surface becomes infeasible and the use of local reconstruction techniques, like splatting [LMR07, PzvBG00, RL00] or implicit reconstruction [AA03, ABCO+03, GG07], has become state of the art. However, these approaches still need some preprocessing steps, constricting instant preview of generated data. With the advent of image-space point-cloud rendering techniques [DRL10, MKC07, RL08, SMK07] it became possible to interactively render and explore scanned datasets on the fly without any preprocessing.

These new possibilities open up a whole set of new applications, but also induce new challenges. The sampling of the generated point clouds can be highly varying, making the use of several rendering approaches difficult. This can be circumvented by using level-of-detail methods conveying a nearly uniform sampling in image space [BWK02, GZPG10, RD10].

Registration of color scans, produced under different light conditions, can result in luminance inconsistencies of the resulting colored point cloud. Consequently, renderings of such point clouds exhibit significant high-frequency noise. Removing such noise has always been an important topic in image processing. There exists a vast amount of approaches in this field [BC04, BJ10, KS10, WWPS10], which typically try to remove noise in an image by analyzing the spatial neighborhood of a pixel and adjusting the pixel value accordingly. Adams et al. [AGDL09] propose a *k*d-tree-based filtering which is also able to handle geometry if connectivity information is given. This is not the case for point clouds resulting from standard scanning techniques. The aforementioned approaches are specialized on denoising images and do not utilize the particular nature of point-cloud renderings. A notable example of denoising that was explicitly designed for point clouds was presented by Kawata and Kanai [KK05], but it suffers from the restriction to only two different points for denoising.

We will show how to effectively exploit the whole amount of surface points that project to a pixel for interactively generating smooth renderings of inconsistently lit point clouds.

3. GENERAL APPROACH

Let P be a colored point cloud, i.e. a finite set of points $\mathbf{p} \in \mathbb{R}^3$ where each point is enhanced with RGB color information. Furthermore, we assume that colors stored at the surface points approximate the respective surface color except for luminance correctly. Our goal is to produce an interactive rendering of the point cloud with smoothly varying luminance, following the assumption that neighboring points represent surface parts with similar luminance.

To adjust the luminance of the point cloud we decided to use the HSV color model, since it naturally describes the luminance of a point $\mathbf{p}$ in its V component. Thus, we are able to manipulate luminance without interfering with hue and saturation. The basic approach is to convert the colors of all points to the HSV model, average the V component between selected surface points and convert the colors back to the RGB format for final rendering.

As a first idea one could think of prefiltering the whole point cloud in object space to achieve this goal. We implement this idea by generating a space partition for the point cloud, enabling the efficient generation of neighborhood information. Luminance of neighboring surface points is smoothed to generate a point cloud with smoothly varying luminance, see Section 4. This approach can effectively eliminate the luminance noise in point clouds when choosing a sufficiently large neighborhood. However, it takes a significant amount of precomputation time, especially for massive point clouds with hundreds of million

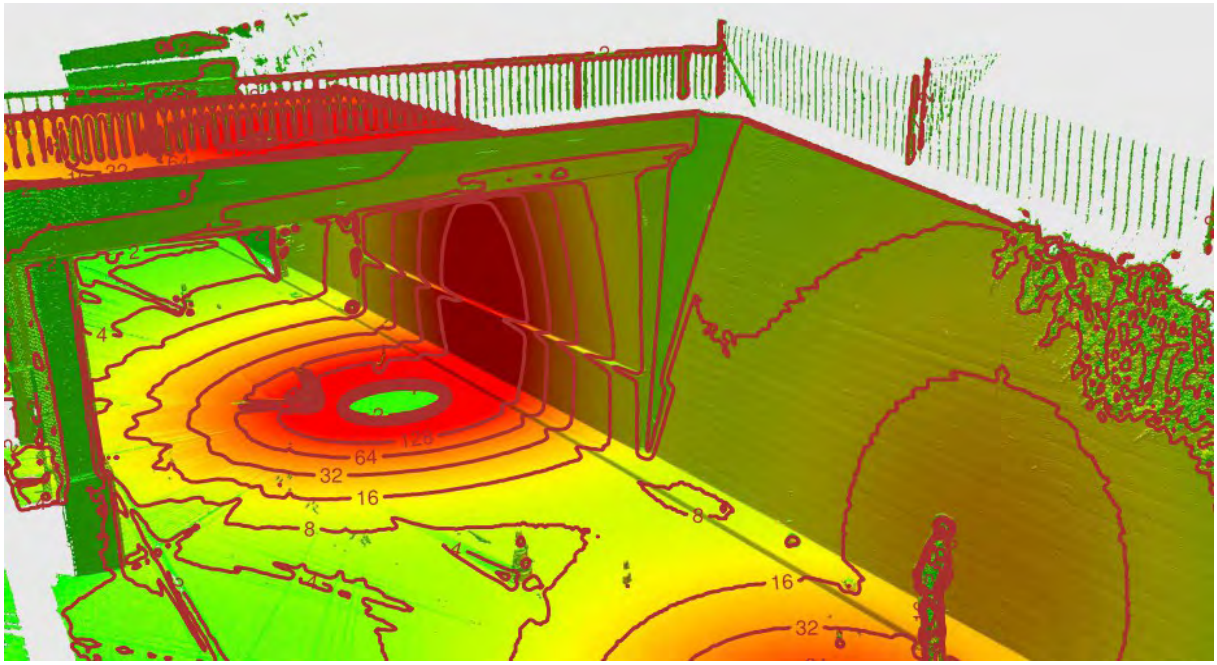


Figure 1. Image-space rendering of a real world point cloud. The number of projected points per pixel is color coded between 1 (green) and 150 (red), which is also emphasized by the isolines. The image was produced with applied depth thresholding and shading to enhance geometry perception.

points, which inhibits the instant rendering of generated point clouds.

To avoid this preprocessing step, we propose a GPU-assisted image-space luminance correction working on the fly. The approach utilizes the fact, that in most cases multiple surface points get projected to one pixel during rendering, as illustrated in Figure 1. When restricting the surface points to those representing non-occluded surface parts, a good approximation for the desired luminance can be obtained by averaging the luminance of the respective surface points. This is done in two rendering steps. In a first pass the scene is rendered to the depth buffer generating a depth mask. Following the idea of a soft z-buffer [PCD+97], an additional threshold ϵ is added to the depth mask, which defines the minimal distance between different consecutive surfaces. In a second render pass the depth mask is utilized to accumulate the luminance of all surface points, effectively contributing to a pixel. A detailed description of the method is given in Section 5.

After this step we apply image-space filters to fill pixels incorrectly displaying background color or occluded surface parts, as proposed by Rosenthal and Linsen [RL08]. The final rendering with associated depth buffer can be used to approximate surface normals per fragment, which opens up several possibilities for calculating postprocessing effects.

4. OFFLINE LUMINANCE CORRECTION

Luminance correction in object space requires the definition of a certain neighborhood for each surface point. We use the n nearest neighbors for each point as neighborhood. For fast detection of these neighborhoods, a three-dimensional kd-tree is built for the point cloud. Since luminance correction is done utilizing the HSV color space, all point colors are converted to this space. Then for each point we compute its neighbors and average their luminance. Finally the complete point cloud is converted back to RGB for rendering.

Note, that for weighted averaging also a kernel function can be used. However, several tests, e.g. with a Gaussian kernel, revealed no significant differences. Regarding the neighborhood size a value of $n=40$ has proven to produce appealing results. However, the precomputation time increases significantly with the number of points and number of neighbors. The luminance correction of a point cloud with 150 million surface points takes for example nearly six hours in an out-of-core implementation. Also when using an in-core implementation, the computation times are far too long for allowing instant views of generated point clouds.

5. IMAGE-SPACE LUMINANCE CORRECTION

Following the main idea of image-space point-cloud rendering, we propose an algorithm that facilitates high-quality point-cloud inspection without preprocessing, utilizing luminance correction in image space. The algorithm takes advantage of the fact that many points are projected to one pixel in models with high point densities, as already shown in Figure 1. Usually a large fraction of these points describe nearly the same geometric position on the surface. The other points belong to parts of the scene which are occluded by the surface closest to the viewer.

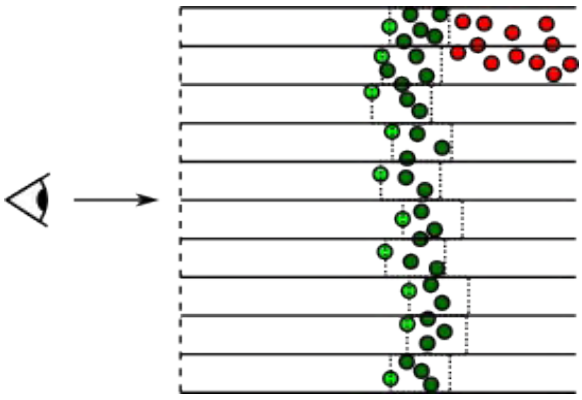
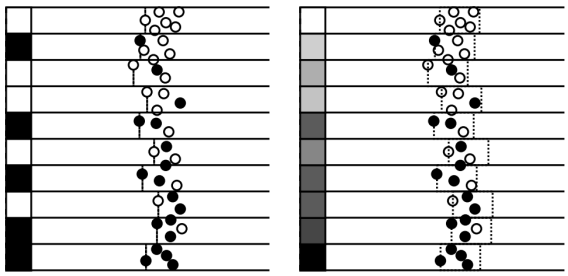


Figure 2. 2D illustration of point projection. The traditional z-buffer test renders only the nearest points (light green). Points representing occluded surface parts (red) but also redundant points on the same surface (dark green) are discarded. By increasing a fragment's z-value by a threshold ϵ (dotted lines) we still discard points from occluded surface parts but are able to blend the luminance of the remaining points for each pixel. (The view plane is the dashed line on the left-hand side.)

Our algorithm for correcting luminance in image space consists of two main rendering steps. In a first rendering pass a (linearized) depth map, selecting all points which represent visible surface parts, is generated (see Figure 2). In a second pass luminance is corrected for each pixel, taking all surface points into account which pass the depth-mask test.

For the first rendering pass all points are rendered resulting in a preliminary depth map and a texture T with the preliminary rendering result. The depth map is extended fragment-wise by the z-threshold, generating the desired, slightly displaced depth mask. Afterwards, we prohibit writing operations to the depth buffer such that every surface point that is closer than the value stored in the depth mask is drawn in the next render pass while farther points are discarded.



a) **b)**

Figure 3. Comparison of the rendering results with (a) normal z-buffering and (b) depth masked luminance correction. The change of predominant luminance at the surface points from top to bottom is rendered much smoother with luminance correction.

In the second render pass we accumulate the luminance of the rendered points using the OpenGL blending with a strictly additive blending function. All surface points are rendered to a non-clamped RGB floating point texture using the depth test. In the fragment shader we set the color of each fragment to (luminance,0,1), which produces a texture $\tilde{T}$ with the accumulated luminances in the R component and the number of blended points in the B component. Finally, we combine the blended texture $\tilde{T}$ with the preliminary rendering result T by converting T to the HSV color space, applying

$$T_{HSV} := (T_H, T_S, \frac{\tilde{T}_R}{\tilde{T}_B})$$

and converting the result back to the RGB color space. The result per pixel is a color with averaged luminance over all surface points, which passed the depth test, i.e. which belong to the visible surface.

Note that one can also easily average colors in RGB space by using a four-channel texture for $\tilde{T}$ and blending (R,G,B,1) for each fragment. Then the resulting color would be given by

$$T_{RGB} := (\frac{\tilde{T}_R}{\tilde{T}_A}, \frac{\tilde{T}_G}{\tilde{T}_A}, \frac{\tilde{T}_B}{\tilde{T}_A})$$

The difference between traditional z-buffering and our approach is depicted in Figure 3. Although our algorithm requires two rendering passes and therefore basically halves the framerate, we are able to produce smooth lighting much faster than with the preprocessing algorithm, making on-site preview feasible.

An enhancement to the base algorithm is to correct luminance not only in one pixel but to additionally use points from neighboring pixels. We do this by summing the luminance of a fragment in the final step over its 8-neighborhood and dividing by the total number of points. A Gaussian kernel can be used as additional weight for the points of the neighboring fragments to take their distances into account. This produces even smoother results than the simple

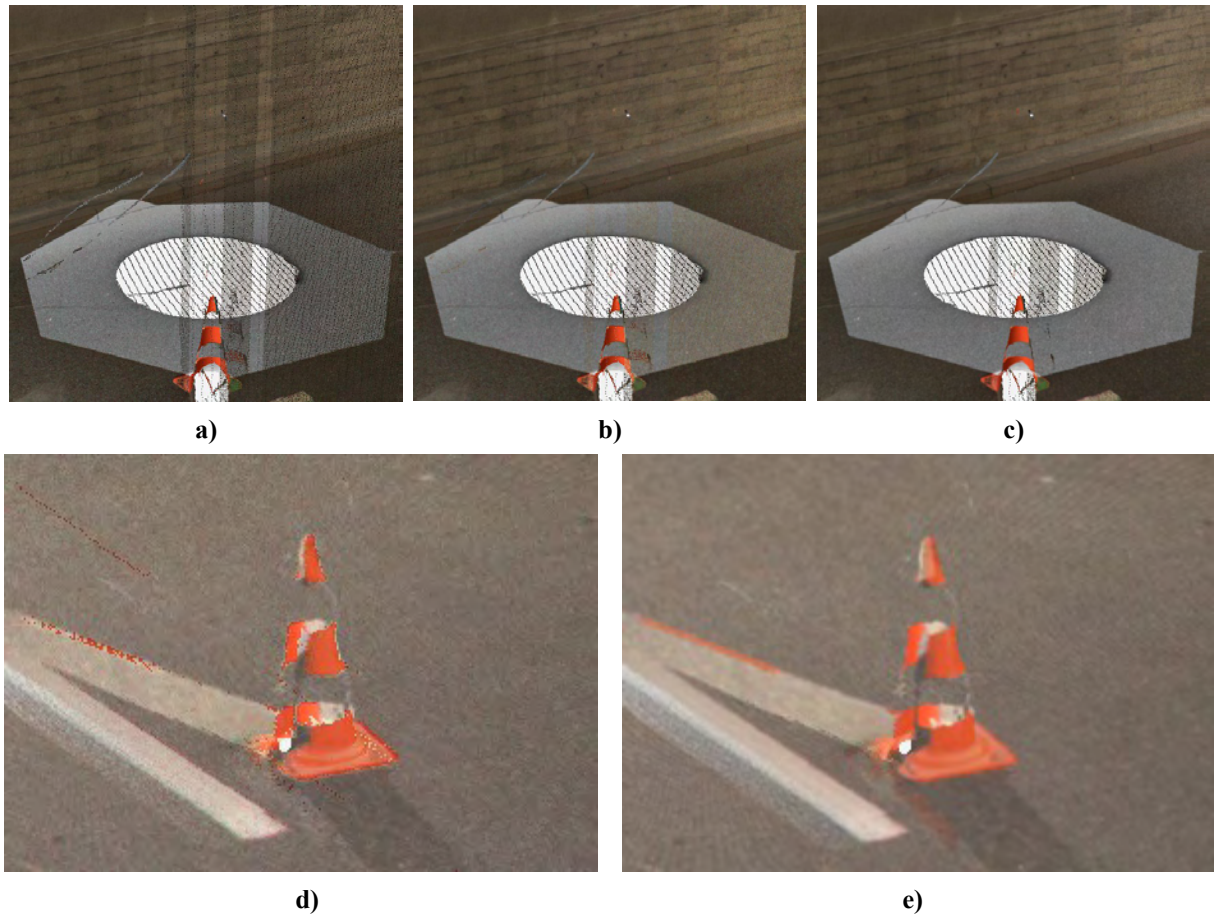


Figure 4. (a) - (c) Comparison of averaging in HSV colorspace without (left) and with Gaussian neighborhood (middle) and RGB space with Gaussian neighborhood (right). A part of the wall is visible between the viewer and the street. While in HSV space the wall is only brightened up but still visible, in RGB space it is smoothed away. (d) - (e) HSV smoothing (left) preserves sharp corners and the structure of the road in the rendering while using RGB (right) produces an antialiased image. In both cases the 3×3 neighborhood has been used.

averaging per fragment while not overblurring the image.

The advantage of using the HSV space lies in the error tolerance when applying the Gaussian. While RGB averaging produces smoother transitions in the image, it tends to blur away details in the rendering. HSV averaging in contrast preserves even small features and sharp edges if this is desired (Figure 4).

The amount of points necessary to successfully use this approach can be roughly approximated by the following calculation: Assume that a spherical volume element with a volume of $1\,\mathrm{cm}^3$ is projected to the center of a Full HD screen. Then the volume element gets projected to a spherical disc with the absolute (world-space) radius $r = \sqrt[3]{\frac{3}{4\pi}}$. Now let α be the vertical field of view of the virtual camera. The projected area of the volume element in distance d from the camera is then given by

$$A = \pi \cdot \left(\frac{1080\, r}{2 d \tan(\alpha)} \right)^2 .$$

Our experiments have shown that a number of seven to ten points per pixel usually suffice to yield smooth transitions. Therefore, in order to view a model from a given distance d, the resolution k of the dataset in terms of points per cm^3 has to satisfy $k \geq 10 \cdot A$, which corresponds to around 50 points per cm^3 for a view from a distance of 5 meters with $\alpha = 35°$. If this requirement is met the angle from which the point cloud is viewed is not of great importance since the geometric measuring accuracy is usually very high, producing only minor inconsistencies in the point-to-pixel mapping in adjacent views. It can only pose problems in the immediate vicinity of edges, when the depth threshold is so high that occluded points are included during blending. This, however, can be mitigated by a sufficiently low threshold and did not induce visible errors in our experiments.

Having normalized the luminance of our points we can fill remaining small holes in the rendering using the image-space filters proposed by Rosenthal and Linsen [RL08]. Finally we apply a simple triangulation scheme over a pixel's 8-connected neighborhood in image space to achieve a satisfying rendering enhanced with lighting and ambient occlusion (See Figure 8).

6. RESULTS AND DISCUSSION

We have tested our luminance correction approach in terms of quality and speed with the help of two types of datasets: unaltered real world data (Hinwil) and real world data with artificial noise (Lucy, dataset courtesy of the Stanford 3D Scanning Repository). We implemented the method using C++ and GLSL on an Intel Core i7-970 machine with an NVIDIA Quadro 4000 graphics card. All results were obtained by rendering to a viewport of 1920 × 1080 pixels.

The Hinwil dataset was acquired by scanning a bridge with several stationary high-resolution laser scanners. Two directly adjacent scans, each one containing about 30 million points, were registered using a geometry-based approach to obtain a dataset with 59 million points. The two scans were taken at different times of the day, resulting in different lighting conditions. An image-space rendering of the data is shown in Figure 6. It exhibits three common problems in such data that are highlighted and magnified:

1. Scattering: A part of the object was scanned from different scanners, but the point cloud from one scanner is significantly more dense in that region than the other's. The result is a noisy appearance of the surface because single points are scattered over the surface.
2. Shadowing: A part of the object can be seen from only one scanner, resulting in aliased lighting borders.
3. Border regions: These are the regions, where the point density of two scanners is nearly equal, causing sharp borders when applying a median filter.

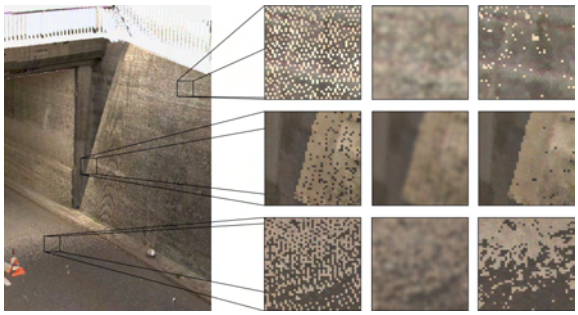
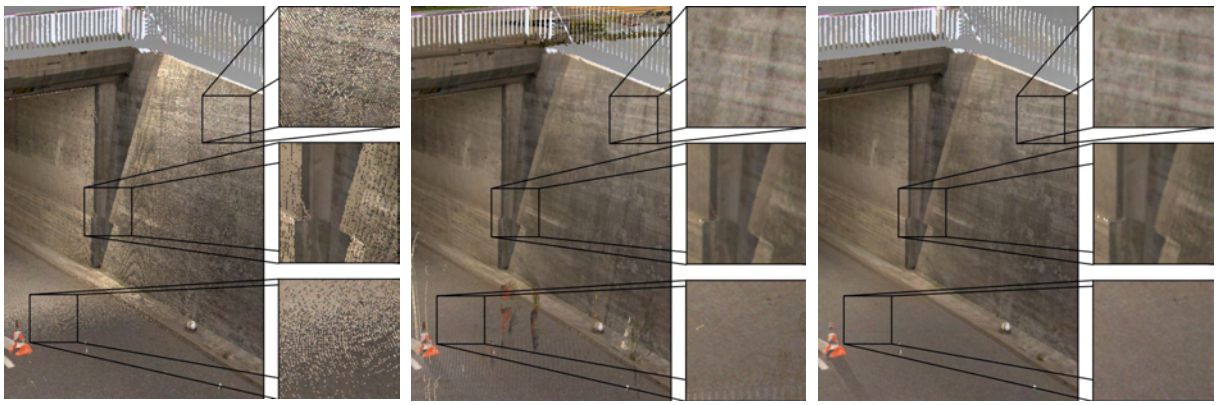


Figure 5. Image-space point-cloud rendering of the Hinwil dataset with closeup views of three problematic regions (left column). The application of standard image filters, like a Gaussian filter (middle column) or a median filter (right column) is not producing satisfying results.

The first problem can, in many cases, be satisfyingly handled by a median filter, which fails at the border regions since it produces a sharp border between the scans. Smoothing with a Gaussian filter yields slightly better results in border regions, but it keeps most of the scattered specks and leads to an overall blurry appearance of the rendered image, as illustrated in Figure 5. The shadowing problem is not easy to solve in image space, since we would have to correct the lighting over larger planar parts of the object.



a) **b)** **c)**

Figure 6. Detail view of the Hinwil dataset using different approaches. For each approach, the overview and three closeup views are presented. (a) Image-space point-cloud rendering, exhibiting typical artifacts in regions with inconsistent lighting. (b) Image-space point-cloud rendering of the complete dataset with offline luminance correction (n=50). The preprocessing effectively eliminates the artifacts. (c) Image-space point-cloud rendering with image-space luminance correction. The visual quality is comparable to the offline method.

a) **b)**

Figure 7. An image-space rendering of the Lucy model with artificial noise (a) without and (b) with luminance correction. The model was artificially up-sampled to 40 million points to achieve a high-enough point density. The normalization was again carried out over the weighted neighborhood. In this image the prevailing noise is less visible than in the torus rendering, since the surface shape of the model is not as homogeneous as the torus.

# Points	Rendering	Rendering + Correction
1M	140 fps	80 fps
4M	75 fps	40 fps
16M	19 fps	10 fps
64M	5 fps	2.5 fps

Table 1. Performance of luminance correction. For different numbers of points the performance for just image-space point-cloud rendering and for the combination with image-space luminance correction is given in frames per second.

Our image-space approach eliminates most of these problems and even weakens the sharp shadow borders slightly. In Figure 6(c) a smooth transition from one scan to the other was achieved without emphasizing borders or blurring the image. To judge our approach in terms of quality, we compare the result to the one obtained by offline luminance correction, shown in Figure 6(b). Both approaches achieve similar results in terms of quality. However, the image-space luminance correction is able to interactively display the dataset without time-consuming preprocessing (offline luminance correction took more than one hour of computation time).

To evaluate the performance of our image-space algorithm we used the full Hinwil dataset with 138 million surface points and downsampled it to different resolutions. As shown in Table 1 the average rendering performance decreases by around 45% when applying image-space luminance correction. This is due to the two-pass nature of our approach.

As a real world dataset with artificial luminance noise we used the Lucy dataset, which consists of 14 million surface points. To achieve sufficient point density in close up views we upsampled the model to 40 million points and, since we were only interested in point data, we discarded the given connectivity information and colored each point uniformly white. We simulated the effects of scans under different lighting conditions by shading the surfaces points using the Phong model with a light source randomly positioned on a 90° circular arc directly above the model. An image-space rendering without luminance correction as well as an image-space rendering with

Figure 8. Rendering of a scene with image-space luminance correction, lighting and screen space ambient occlusion.

image-space luminance correction are shown in Figure 7. Our algorithm was able to largely remove the noise and yielded a satisfying rendering.

7. CONCLUSIONS

We have presented an approach for rendering point clouds with corrected luminance value at interactive frame rates. Our luminance correction operates in image space and is applied on the fly. As such, we have achieved our goal to allow for instantaneous rendering of large point clouds taken from multiple 3D scans of architecture. No preprocessing is necessary and the quality of the results is pleasing.

In order to work properly our algorithm relies on a sufficient number of points per pixel. Moreover, we observed in our experiments that single very bright scanlines from far away scanners were blended with darker regions of small point density, still leading to noticeable artifacts. This can be solved by considering only point clouds from directly adjacent scanners for blending which, at the present time, was done manually but will hopefully be automated in a future version.

8. ACKNOWLEDGMENTS

The authors would like to thank the enertec engineering AG (Winterthur, Switzerland) for providing us with the real-world data and for their close collaboration. This work was partially funded by EUREKA Eurostars (Project E!7001 "enercloud").

9. REFERENCES

[AA03] Anders Adamson and Marc Alexa. Ray tracing point set surfaces. In SMI '03: Proceedings of the Shape Modeling International 2003, pp.272–279, Washington, DC, USA, 2003. IEEE Computer Society.

[ABCO+03] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Computing and rendering point set surfaces. IEEE Transactions on Visualization and Computer Graphics, 9:3–15, 2003.

[AGDL09] Andrew Adams, Natasha Gelfand, Jennifer Dolson, and Marc Levoy. Gaussian kd-trees for fast high-dimensional filtering. ACM Transactions on Graphics, 28:21–33, 2009.

[BC04] Danny Barash and Dorin Comaniciu. A common framework for nonlinear diffusion, adaptive smoothing, bilateral filtering and mean shift. Image and Vision Computing, 22(1):73 – 81, 2004.

[BJ10] Jongmin Baek and David E. Jacobs. Accelerating spatially varying gaussian filters. ACM Transactions on Graphics, 29:169–179, 2010.

[BR02] Fausto Bernardini and Holly Rushmeier. The 3d model acquisition pipeline. Computer Graphics Forum, 21(2):149–172, 2002.

[BWK02] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In Proceedings of the 13th Eurographics workshop on Rendering, pp.53–64, 2002.

[DRL10] Petar Dobrev, Paul Rosenthal, and Lars Linsen. Interactive image-space point cloud rendering with

transparency and shadows. In Vaclav Skala, editor, Communication Papers Proceedings of WSCG, The 18th International Conference on Computer Graphics, Visualization and Computer Vision, pp.101–108, Plzen, Czech Republic, 2010. UNION Agency – Science Press.

[GG07] Gaël Guennebaud and Markus Gross. Algebraic point set surfaces. ACM Transactions on Graphics, 26, 2007.

[GZPG10] P. Goswami, Y. Zhang, R. Pajarola, and E. Gobbetti. High quality interactive rendering of massive point models using multi-way kd-Trees. In Pacific Graphics Poster Papers, 2010.

[KK05] Hiroaki Kawata and Takashi Kanai. Direct point rendering on gpu. In George Bebis, Richard Boyle, Darko Koracin, and Bahram Parvin, editors, Advances in Visual Computing, volume 3804 of Lecture Notes in Computer Science, pp.587–594. Springer Berlin / Heidelberg, 2005.

[KS10] Michael Kass and Justin Solomon. Smoothed local histogram filters. ACM Transactions on Graphics, 29:100–110, 2010.

[LMR07] Lars Linsen, Karsten Müller, and Paul Rosenthal. Splat-based ray tracing of point clouds. Journal of WSCG, 15:51–58, 2007.

[LPC+00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. In Proceedings of the 27th annual conference on Computer graphics and interactive techniques, SIGGRAPH '00, pp.131–144, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[MKC07] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. Efficient point-based rendering using image reconstruction. In Proceedings of the Symposium on Point-Based Graphics, pp.101–108, 2007.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: surface elements as rendering primitives. In SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp.335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[RD10] R. Richter and J. Döllner. Out-of-core real-time visualization of massive 3D point clouds. In Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, pp.121–128, 2010.

[RL00] S. Rusinkiewicz and M. Levoy. QSplat: a multiresolution point rendering system for large meshes. In Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp.343–352, 2000.

[RL08] Paul Rosenthal and Lars Linsen. Image-space point cloud rendering. In Proceedings of Computer Graphics International, pp.136–143, 2008.

[SMK07] R. Schnabel, S. Moeser, and R. Klein. A parallelly decodeable compression scheme for efficient point-cloud rendering. In Proceedings of Symposium on Point-Based Graphics, pp.214–226, 2007.

[SSS06] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: exploring photo collections in 3D. ACM Transactions on Graphics, 25:835–846, 2006.

[TS08] Thorsten Thormählen and Hans-Peter Seidel. 3D-modeling by ortho-image generation from image sequences. ACM Transactions on Graphics, 27:86:1–86:5, 2008.

[WBB+08] Michael Wand, Alexander Berner, Martin Bokeloh, Philipp Jenke, Arno Fleck, Mark Hoffmann, Benjamin Maier, Dirk Staneker, Andreas Schilling, and Hans-Peter Seidel. Processing and interactive editing of huge point clouds from 3D scanners. Computers & Graphics, 32(2):204–220, 2008.

[WWPS10] Z. Wang, L. Wang, Y. Peng, and I. . Shen. Edge-preserving based adaptive icc method for image diffusion. In Proceedings of the 3rd International Congress on Image and Signal Processing, CISP 2010, volume 4, pp.1638–1641, 2010.

Subpixel Reconstruction Antialiasing for Ray Tracing

Chiu, Y.-F
National Tsing Hua University, Taiwan
yfchiu@ibr.cs.nthu.edu.tw

Chen, Y.-C
National Tsing Hua University, Taiwan
louis@ibr.cs.nthu.edu.tw

Chang, C.-F
National Taiwan Normal University, Taiwan
chunfa@ntnu.edu.tw

Lee, R.-R
National Tsing Hua University, Taiwan
rrlee@cs.nthu.edu.tw

ABSTRACT

We introduce a practical antialiasing approach for interactive ray tracing and path tracing. Our method is inspired by the Subpixel Reconstruction Antialiasing (SRAA) method which separates the shading from visibility and geometry sampling to produce antialiased images at reduced cost. While SRAA is designed for GPU-based deferred shading renderer, we extend the concept to ray-tracing based applications. We take a hybrid rendering approach in which we add a GPU rasterization step to produce the depth and normal buffers with subpixel resolution. By utilizing those extra buffers, we are able to produce antialiased ray traced images without incurring performance penalty of tracing additional primary rays. Furthermore, we go beyond the primary rays and achieve antialiasing for shadow rays and reflective rays as well.

Keywords: antialiasing, ray tracing, path tracing.

1 INTRODUCTION

With the abundance of computation power and parallelism in multicore microprocessors (CPU) and graphics processors (GPU), achieving interactive photorealistic rendering on personal computers is no longer a fantasy. Recently, we have seen the demonstration of real-time ray tracing [6, 17] and the emergence of real-time path tracing with sophisticated global illumination [2, 20]. Though real-time path tracing can produce rendering of photorealistic quality that include complex lighting effects such as indirect lighting and soft shadow, the illusion of a photograph-like image breaks down quickly when jaggy edges are visible (Figure 1 shows an example).

Jaggy edges are one of the typical aliasing artifacts in computer generated images. A straightforward antialiasing technique is to increase the sampling rate by taking multiple samples uniformly at various subpixel positions. However this approach induces significant performance penalty that makes it an afterthought in real-time ray tracing. A more practical approach is to increase subpixel samples adaptively for image pixels where discontinuity is detected. Although adaptive sampling approach avoids the huge performance hit of the multisampling approach, it still requires additional subpixel samples and introduces large variation to the estimation of rendering time.

In this work, we introduce an antialiasing approach that works well for real-time ray tracing and path tracing. We take a hybrid rendering approach in which we add a GPU rasterization step to produce the depth and normal buffers with subpixel resolution. By utilizing those extra buffers, we are able to produce antialiased ray traced images without incurring performance penalty of tracing additional primary rays. Our method is inspired by the Subpixel Reconstruction Antialiasing (SRAA) [3] which combines per-pixel shading with subpixel visibility to produce antialiased images. While SRAA is designed for GPU-based deferred shading renderer, we extend the concept to ray-tracing based applications. Furthermore, we apply our antialiasing approach to shadow and reflection which SRAA cannot resolve with its subpixel buffers.

Our main contributions in this work are:

- We propose an efficient antialiasing technique which improves the perception of photorealism in interactive or real-time ray tracing without sacrificing its performance.
- Unlike adaptive sampling or subpixel sampling, our approach does not penalize the performance of a CPU ray tracer because no additional primary ray needs to be traced. Our hybrid rendering approach obtains the necessary subpixel geometric information by leveraging the GPU rasterization pipeline.
- While SRAA works well for improving the sampling on image plane, we extend its application beyond the primary rays and achieve antialiasing for shadow rays and reflective rays as well.

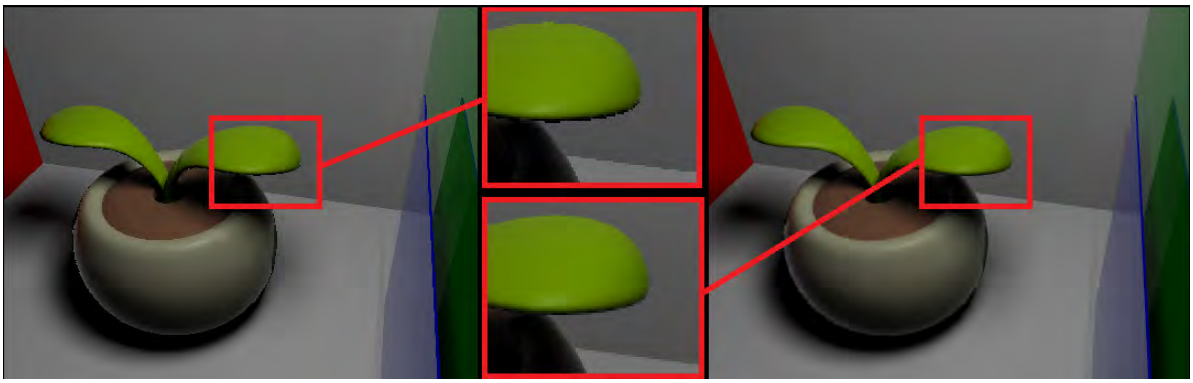


Figure 1: Antialiasing can improve the rendering quality in interactive ray tracing. The left image is rendered without applying any antialiasing method. The right image is rendered with our method using relatively inexpensive geometric information to improve expensive shading results.

2 RELATED WORK

2.1 Real-time Ray Tracing

With the rapid improvement of computation power and parallelism in multicore microprocessors (CPU) and graphics processors (GPU), various works have been published on speeding up the ray tracing, either on CPUs [1, 19], GPUs [5], or special-purpose platforms [21]. Recently, we have seen the demonstration of real-time ray tracing with Whitted-style reflection and refraction (a.k.a. specular rays) [6, 17] and the emergence of real-time path tracing with sophisticated global illumination [2, 20]. The NVIDIA OptiX acceleration engine [12] elevates rendering applications to a new level of interactive realism by greatly increasing ray tracing speeds with GPU solutions. While real-time ray tracing is now feasible, most renderers still rely on Monte Carlo path tracing to obtain more sophisticated global illumination effects such as soft shadow and indirect lighting. Noisy preview images are usually produced first at interactive rates and then gradually converge to high quality images. Therefore, antialiasing often becomes an afterthought as it further slows down the rendering.

2.2 Adaptive Sampling

The ray tracing algorithm is basically a loop over all screen pixels to find the nearest visible object in the scene. We can consider ray tracing as a point sampling based rendering method in signal processing view. However, point sampling makes an all-or-nothing choice in each pixel and thus leads to jaggies. Antialiasing of ray-traced images could be achieved by supersampling the image. However the supersampling approach demands significantly larger amount of computation resource. Therefore antialiasing by supersampling is rarely adopted by software renderers. Adaptive sampling [10, 11] reduces the overhead by casting additional rays only if significant color variation across image samples is detected. Variations of the adaptive sampling techniques have also been proposed in [8].

2.3 Post Filter Antialiasing

A small disadvantage of adaptive sampling is that some image pixels still need additional subpixel samples to be fully shaded or traced. It would be desirable if expensive shading could be avoided at additional subpixel locations. Reshetov [16] proposes an image filtering approach, Morphological antialiasing (MLAA) to recover edges from input image with per-pixel color information. However, this sort of color-only information could fail to identify some geometry edges, especially those edges without high contrast. Geometric Post-process Anti-Aliasing (GPAA) [13] and Geometry Buffer Anti-Aliasing (GBAA) [14] extend the MLAA ideas and use extra edge information explicitly to eliminate the jaggy edges. Normal Filter Anti-Aliasing (NFAA) [18] reduces aliasing by searching for contrasting luminosity changes in the final rendering image. It builds a normal displacement map to apply a per-pixel blur filter in highly contrast aliased areas. However, it softens the image due to the filtering of textures. More filter-based approaches are discussed in [7].

2.4 Shading Reconstruction Filter

Decoupled sampling [9, 15] presents an approach to generate shading and visibility samples at different rates in GPU pipelines to speed up the rendering in applications with stochastic supersampling, depth of field, and motion blur. Yang et al. [22] present a geometry-aware framebuffer level of detail (LOD) approach for controlling the pixel workload by rendering a subsampled image and using edge-preserving upsampling to the final resolution. Subpixel Reconstruction Antialiasing (SRAA) [3] takes a similar decoupled sampling approach and applies a cross-bilateral filter (as in the geometry-aware framebuffer LOD method) to

upscale shading information using subpixel geometric information that is obtained from the GPU rasterization pipeline. It is based on the assumption that the subpixel geometric information could be obtained much more easily without fully going through the expensive shading stage. SRAA can produce good edge antialiasing but it cannot resolve shading edges in texture, shadow, reflection and refraction. Our work follows the same assumption by avoiding emitting subpixel samples for the primary rays. This maintains the advantage over adaptive sampling because no subpixel ray needs to be traced.

3 ANTIALIASING

SRAA [3] relies on the fact that shading often changes more slowly than geometry in screen space and generates shading and visibility at different rates. SRAA performs high-quality antialiasing in a deferred rendering framework by sampling geometry at higher resolution than the shaded pixels. It makes three modifications to a standard rendering pipeline. First, it must produce normal and depth information at subpixel resolution. Second, it needs to reconstruct the shading values of sampled geometric subpixel from neighboring shaded samples with bilateral filter using the subpixel geometric (normal and depth) information. Finally, the subpixel shading values are filtered into an antialiased screen-resolution image.

SRAA detects the geometric edges with geometric information to resolve aliasing problem. However, the edges of shadow and reflection/refraction could not be detected by the subpixel geometric information generated from the eye position. For example, the shadow edges mostly fall on other continuous surfaces that have slowly changing subpixel depths and normals. To extend the SRAA concept to ray-tracing based applications, we perform antialiasing separately for primary rays, shadow rays and secondary rays to resolve this issue. The following subsections offer the detail.

3.1 Primary Ray

Like SRAA, our goal is to avoid the performance penalty of shading subpixel samples. In Figure 2, geometric information and shading are generated at different rates. Each pixel has 4 geometric samples on a 4×4 grid and one of those geometric samples is also a shaded sample. The shading value at each geometric sample is reconstructed by interpolating all shaded neighbors in a fixed radius using the bilateral weights. We take both depth and normal change into account when compute the bilateral weight. A neighboring sample with significantly different geometry is probably across a geometric edge and hence receives a low weight.

$$w_{ij} = G(\sigma_z(z_j - z_i))G(\sigma_n(1 - sat(n_j \cdot n_i))) \quad (1)$$

In Equation 1, $G(x)$ is the Gaussian function of the form $exp(-x^2)$. z_i and n_i are the depth and normal of the i^{th} subpixel sample. σ_z and σ_n are the scaling factors for controlling how quickly the weights fall off and allowing us to increase the importance of the bilateral filter. We set σ_z to 10 and σ_n to 0.25 in all our testing. The $sat(x)$ function is implemented as $max(0, min(1, x))$. The result w_{ij} is the weight associated with the j^{th} subpixel sample while performing shading reconstruction for the i^{th} subpixel sample.

For tracing the primary rays that are emitted from the eye position, we use a hybrid rendering approach that utilizes the GPU to generate the subpixel geometric information including position, normal and depth. We create 3 auxiliary geometric buffers to store position, normal and depth by GPU rasterization with the same resolution as the shaded buffer. Each geometric buffer is rendered with a subpixel offset applied to the projection matrix. The subpixel offset is applied not only to form a $4\times$ rotated-grid but also to do pixel alignment between rasterization and ray tracing rendering. Since the GPU rasterization pipeline produces the subpixel geometric information very efficiently, this overhead is insignificant when compared to the ray tracing stage.

3.2 Shadow Ray

As mentioned above in Section 3, the shadow edges cannot be detected by the geometric information that is generated from the eye position alone. What we need is subpixel information that is more meaningful to the shadow edges. The naive solution for shadow antialiasing is through a shadow map drawn at a higher resolution. However, this approach is inefficient because the increased resolution of the shadow map (from the light's view) does not contribute directly to the subpixels at the screen space. Therefore, we generate subpixel shadow information by ray casting and combine this shadow value with the bilateral filter weighting equation as shown in Equation 2. The subpixel shadow rays are generated by utilizing the position information in the geometric buffer as mentioned in Section 3.1.

Figure 2 shows our algorithm reconstructs the color value of a geometric sample in a non-shadowed area not only by taking the Euclidean distance and the normal change between the source and the target samples but also under the influence of shadow boundaries to exclude the neighboring samples in shadowed area. This is the reason why the original SRAA adds excessive blur to the shadow boundaries, yet our method achieves a better quality that is comparable to $16\times$ supersampling.

$$w_{ij} = \begin{cases} G(\sigma_z(z_j - z_i))G(\sigma_n(1 - sat(n_j \cdot n_i))), & \text{if } s_i = s_j \\ 0, & \text{if } s_i \neq s_j \end{cases} \quad (2)$$

In Equation 2, s_i is the shadow value of the i^{th} subpixel sample and is equal to 1 if it is in shadowed area. Otherwise it is equal to 0. If s_j is different from s_i, then the j^{th} subpixel falls on the other side of a shadow edge. Therefore we set the weight w_{ij} associated with the j^{th} subpixel to 0 to exclude it from the shading reconstruction for the i^{th} subpixel sample.

3.3 Secondary Ray

In the original SRAA framework, it uses geometric information to detect geometric edges in the subpixel reconstruction process. However, the edge of secondary shading (such as those from the reflection) cannot be detected by this geometric information generated from the eye position. Take the reflection rays for an example as shown in Figure 5 (c), if we perform subpixel shading reconstruction as shown in Equation 1 with the geometric information generated from the eye position, it will not be able to detect the edges of the reflected objects, and in consequence add excessive blur to the reflected colors.

Therefore we must take geometric information that is generated from the hit points of primary rays to perform subpixel-level bilateral filer when computing the shading value of the secondary rays that originate from the primary hit point. The subpixel secondary rays for hit points are generated by utilizing the position and normal information in the geometric buffer. Our method which performs subpixel reconstruction separately for primary and secondary shading achieves better quality than the original SRAA approach. Please see our results in Section 4 and Figure 5.

4 RESULT

Our algorithm is implemented using NVIDIA CUDA 4.0, the raytracer is built with OptiX 2.0 and rasterization with OpenGL. All results shown in this paper were obtained using an Intel Xeon E5504 processor and an NVIDIA Geforce GTX 570 GPU.

4.1 Quality

Figure 4 shows the quality comparison between our method and other antialiasing techniques in a Cornell box scene. The original SRAA adds excessive blur to the shadow, yet our method achieves similar quality to 16× supersampling.

Figure 5 highlights some interesting cases for primary shading, shadow and secondary shading in the Sponza scene. For the shading from primary rays, both our method and SRAA use the geometric information to improve image quality and the results are almost identical between ours and SRAA. For the shadow, our method uses both the geometry and the shadow edge information to perform subpixel reconstruction, thus produces better shadow line in the highlighted area than SRAA. For secondary shading, we perform subpixel reconstruction separately for primary and secondary shading, while SRAA uses only the final color of each sampled subpixel for this purpose. This results in over blurring for secondary shading in SRAA.

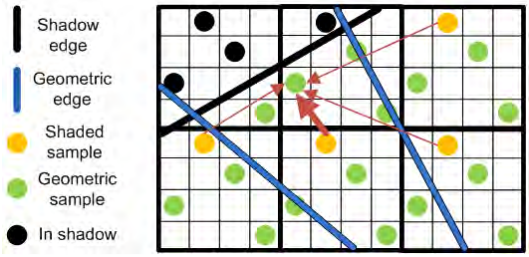


Figure 2: Here we add shadow edge into consideration to perform subpixel shading reconstruction. Each pixel has 4 geometric samples on a 4×4 grid. One of those geometric samples is also a shaded sample. Shading value for each geometric sample in non-shadowed area is reconstructed from nearby shaded samples except the shaded samples in shadowed area and weighted by their distance and the normal change between the source and the target sample.

To summarize, we observe that antialiasing with geometric information from primary rays could be problematic in some difficult cases and our method offers a solution to the highlighted cases in Figure 5.

Our method does have a limitation in handling material variation or textured surfaces. Figure 6 shows such an example where the floor contains patches of different colors. Since the extra subpixel depth and normal information does not help us detect the edges between patches of different colors, jagged edges could still appear on the floor.

4.2 Performance

There are two rendering passes in our current implementation. The first pass is the geometric information generation step and the second pass is the antialiasing process. Table 1 shows that the geometric information generation step with raytracer solution takes about 70 percent of the total processing time for rendering the Sponza scene [4] in Figure 5. This overhead to generate geometric information for primary rays can be reduced with a GPU hybrid solution. Figure 3 shows that our method maintains the interactive rate while rendering the Sponza scene in Figure 5 with a GPU hybrid solu-

	Rendering Pass		
Resolution	1^{st}	2^{nd}	Total
256x256	18	5	23
512x512	35	14	49
768x768	69	28	97
1024x1024	116	49	165

unit: millisecond

Table 1: Time measurement of our method for rendering the Sponza scene in Figure 5.The first pass is geometric information generation and the second pass is antialiasing process. Note that the time shown in first pass is measured with raytracer solution.

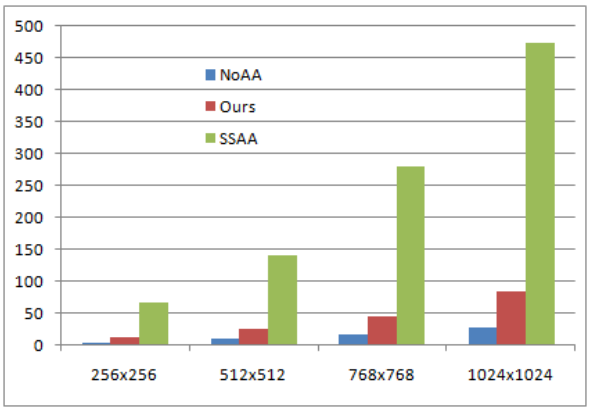


Figure 3: Performance comparison between NoAA (no antialiasing applied), our method with GPU hybrid approach, and SSAA (16× supersampling antialiasing) for rendering the Sponza scene under various output resolutions. The vertical axis is the rendering time in millisecond. The overall rendering performance of our method with a GPU hybrid approach is about 6× speedup in average compared to the 16× supersampling approach.

tion and achieves about 6× speedup in average compared to the 16× supersampling approach.

5 CONCLUSION

We introduce the concept in SRAA to path-tracing based rendering methods for antialiasing. Our method extends the subpixel geometric sampling concept beyond the primary rays and achieves antialiasing for shadow rays and reflective rays as well. By adopting a hybrid approach, our method improves the image quality without incurring performance penalty of tracing additional primary rays. We hope our method encourages the adoption of antialiasing even for the computationally constrained real-time ray tracing or path tracing.

6 ACKNOWLEDGEMENTS

This work is supported in part under the "Embedded software and living service platform and technology development project" of the Institute for Information Industry which is subsidized by the Ministry of Economy Affairs (Taiwan), and by National Science Council (Taiwan) under grant NSC 100-2219-E-003-002.

7 REFERENCES

[1] Carsten Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, 2006.

[2] Jacco Bikker. Arauna real-time ray tracer and Brigade real-time path tracer.

[3] Matthäus G. Chajdas, Morgan McGuire, and David Luebke. Subpixel reconstruction antialiasing for deferred shading. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 15–22, 2011.

[4] Marko Dabrovic. Sponza atrium, http://hdri.cgtechniques.com/ sponza/files/, 2002.

[5] Johannes Gunther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 113–118, 2007.

[6] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, 2007.

[7] Jorge Jimenez, Diego Gutierrez, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, Hugh Malan, Emil Persson, Dmitry Andreev, and Tiago Sousa. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH Courses*, 2011.

[8] Bongjun Jin, Insung Ihm, Byungjoon Chang, Chanmin Park, Wonjong Lee, and Seokyoon Jung. Selective and adaptive supersampling for real-time ray tracing. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 117–125, 2009.

[9] Gábor Liktor and Carsten Dachsbacher. Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 143–150, New York, NY, USA, 2012. ACM.

[10] Don P. Mitchell. Generating antialiased images at low sampling densities. In *Proceedings of the 14th annual conference on Computer graph-*

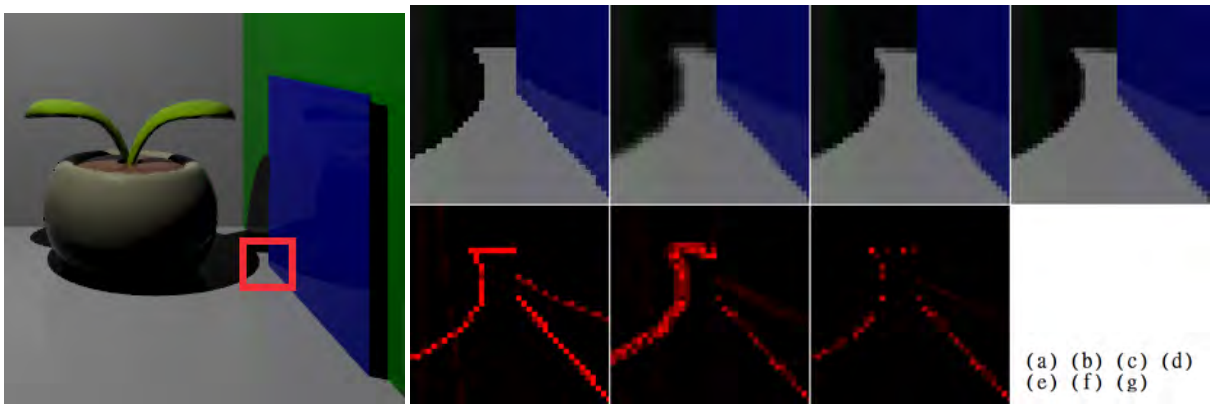


Figure 4: The leftmost image shows the Cornell box generated with our method. The smaller images to its right show the 8× zoom-in of the marked region under various antialiasing techniques. (a) is the result without any antialiasing. (b) is from SRAA. (c) is from our method. (d) is the reference image (16× supersampling antialiasing). (e)(f)(g) show the difference between (a)(b)(c) and the reference image (d) respectively. The original SRAA often adds excessive blur to the shadow and secondary shading, yet our method achieves similar quality to 16× supersampling.

ics and interactive techniques, SIGGRAPH '87, pages 65–72, 1987.

[11] J. Painter and K. Sloan. Antialiased ray tracing by adaptive progressive refinement. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '89, pages 281–288, 1989.

[12] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29:66:1–66:13, July 2010.

[13] Emil "Humus" Persson. Geometric post-process anti-aliasing (GPAA), march 2011, http://www.humus.name/index.php?page=3d&id=86.

[14] Emil "Humus" Persson. Geometry buffer anti-aliasing (GBAA), july 2011, http://www.humus.name/index.php?page=3d&id=87.

[15] Jonathan Ragan-Kelley, Jaakko Lehtinen, Jiawen Chen, Michael Doggett, and Frédo Durand. Decoupled sampling for graphics pipelines. *ACM Trans. Graph.*, 30(3):17:1–17:17, May 2011.

[16] Alexander Reshetov. Morphological antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 109–116, 2009.

[17] Min Shih, Yung-Feng Chiu, Ying-Chieh Chen, and Chun-Fa Chang. Real-time ray tracing with CUDA. In *Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '09, pages 327–337, 2009.

[18] Styves. Normal filter anti-aliasing, http://www.gamedev.net/topic/580517-nfaa—a-post-process-anti-aliasing-filter-results-implementation-details, 2010.

[19] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[20] Sven Woop and Manfred Ernst. Embree - photo-realistic ray tracing kernels, http://software.intel.com/en-us/articles/embree-highly-optimized-visibility-algorithms-for-monte-carlo-ray-tracing/, June 2011.

[21] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24:434–444, July 2005.

[22] Lei Yang, Pedro V. Sander, and Jason Lawrence. Geometry-aware framebuffer level of detail. *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering 2008)*, 27(4):1183–1188, 2008.

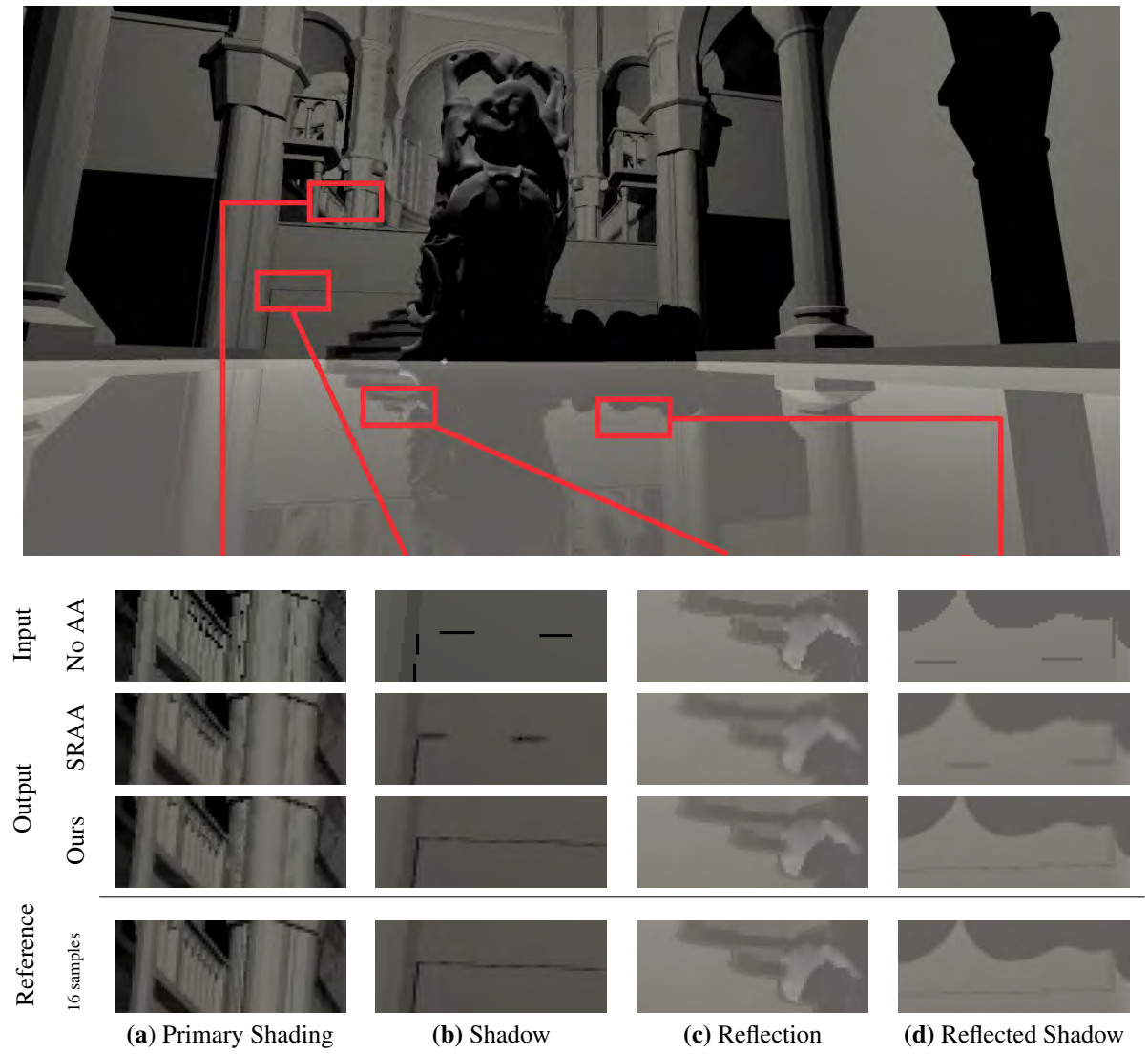


Figure 5: Quality comparison between our method and the other antialiasing techniques in highlighted areas of primary shading, shadow, reflection, and reflected shadow. (Row 1) No antialiasing, (Row 2) SRAA: one subpixel with shading value and 4 subpixels with primary geometric information, (Row 3) Ours: one subpixel with shading value and 4 subpixels with geometric information for primary, shadow and secondary rays, (Row 4) Reference image: 16× supersampling.

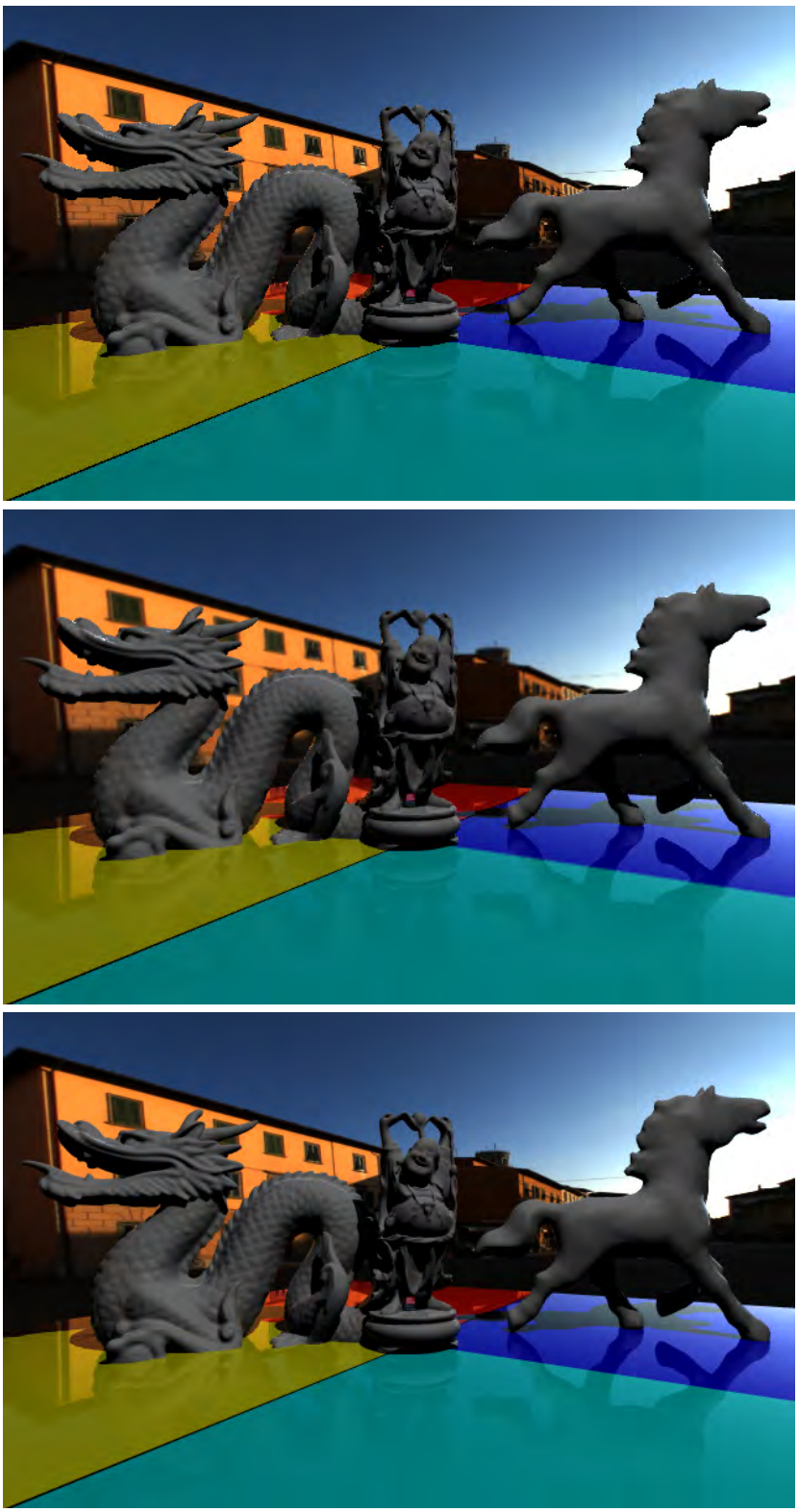


Figure 6: The scene in this figure shows a limitation of our method in handling material variation or textured surfaces. The floor contains patches of different colors. Since the extra subpixel depth and normal information does not help us detect the edges between patches of different colors, jagged edges still appear on the floor in the middle image that is rendered by our antialiasing method. For comparison, the top image shows the result without antialiasing and the bottom image is produced with 16× supersampling.

Elastically Deformable Models based on the Finite Element Method Accelerated on Graphics Hardware using CUDA

Mickeal Verschoor
Eindhoven University of Technology,
The Netherlands
m.verschoor@tue.nl

Andrei C. Jalba
Eindhoven University of
Technology, The Netherlands
a.c.jalba@tue.nl

Abstract

Elastically deformable models have found applications in various areas ranging from mechanical sciences and engineering to computer graphics. The method of Finite Elements has been the tool of choice for solving the underlying PDE, when accuracy and stability of the computations are more important than, e.g., computation time. In this paper we show that the computations involved can be performed very efficiently on modern programmable GPUs, regarded as massively parallel co-processors through Nvidia's CUDA compute paradigm. The resulting global linear system is solved using a highly optimized Conjugate Gradient method. Since the structure of the global sparse matrix does not change during the simulation, its values are updated at each step using the efficient update method proposed in this paper. This allows our fully-fledged FEM-based simulator for elastically deformable models to run at interactive rates. Due to the efficient sparse-matrix update and Conjugate Gradient method, we show that its performance is on par with other state-of-the-art methods, based on e.g. multigrid methods.

Keywords: Elastically deformable models, Finite Elements, sparse-matrix update, GPU.

1 INTRODUCTION

Mathematical and physical modeling of elastically deformable models has been investigated for many years, especially within the fields of material and mechanical sciences, and engineering. In recent years, physically-based modeling has also emerged as an important approach to computer animation and graphics modeling. As nowadays graphics applications demand a growing degree of realism, this poses a number of challenges for the underlying real-time modeling and simulation algorithms. Whereas in engineering applications modeling of deformable objects should be as accurate as possible compared to their physical counterparts, in graphics applications computational efficiency and stability of the simulation have most often the highest priority.

The Finite Element Method (FEM) constitutes one of the most popular approaches in engineering applications which need to solve Partial Differential Equations (PDEs) at high accuracies on irregular grids [PH05]. Accordingly, the (elastically) deformable object is viewed as a continuous connected volume, and the laws of continuum mechanics provide the governing PDE, which is solved using FEM. Other popular methods are the Finite Difference Method (FDM) [TPBF87], the Finite Volume Method (FVM) [TBHF03] and the Boundary Element Method (BEM) [JP99] (see [NMK*06, GM97]). FDM is the easiest to implement, but as it needs regular spatial grids, it is difficult to approximate the boundary of an arbitrary object by a regular mesh. FVM [TBHF03] relies on a geometrical framework, making it more intuitive than FEM. However, it uses heuristics to define the strain tensor and to calculate the force emerging at each node. BEM performs the computations on the surface of the model, thus achieving substantial speedups as the size of the problem is proportional to the area of the model's boundary as opposed to its volume. However, this approach only works for objects whose interior is made of homogeneous material. Furthermore, topological changes are more difficult to handle than in FEM methods [NMK*06].

In this paper we present a fully-fledged FEM-based simulator for elastically-deformable models, running solely on GPU hardware. We show that the involved computations can be performed efficiently on modern programmable GPUs, regarded as massively parallel co-processors through Nvidia's CUDA compute paradigm. Our approach relies on the fast GPU Conjugate Gradient (CG) method of [VJ12] to solve the resulting linear system. Since the topology of the deformed mesh does not change during the simulation, the structure of the sparse-matrix describing the linear system is reused throughout the simulation. However, during the simulation, the matrix values have to be updated efficiently. To achieve this, we propose a method that updates the sparse-matrix entries respecting the ordering of the data, as required by the CG method of [VJ12], see Sect. 5.4. Thanks to the optimized CG method and the efficient sparse-matrix update procedure, we ob-

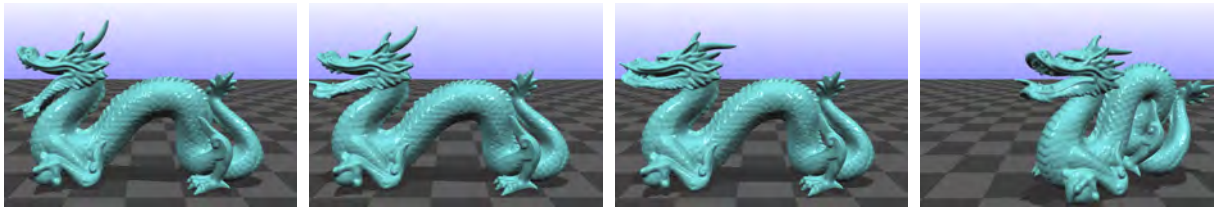


Figure 1: Effect of external (stretching) forces on an 'elastic' dragon.

tain results similar to state-of-the-art multigrid methods [DGW11].

The paper is organized as follows. Sections 3 and 4 describe the involved discretizations using FEM. Next, Section 5 presents the non-trivial parts of our GPU mapping, i.e., computing the local matrices, updating the global sparse matrix and solving the linear system. Finally, in Section 6 results are presented and analyzed.

2 PREVIOUS AND RELATED WORK

Bolz *et al.* [BFGS03], and Krüger and Westermann [KW03] were among the first to implement CG solvers on graphics hardware, using GPU programming based on (fragment) shaders. These authors had to deal with important limitations, *e.g.*, the lack of scatter operations, limited floating-point precision and slow texture switching based on pixel buffers, as exposed by the 'rendering-based' GPU-programming paradigm. One of the first GPU implementations of FEM is due to Rumpf and Strzodka [RS01], in the context of solving linear and anisotropic diffusion equations. Related work on GPU-accelerated FEM simulations also include the papers by Göddeke and collaborators [GST05, GST07, GSMY*07]. However, the emphasis is on improving the accuracy of *scientific* FEM-based simulations. Prior related work with respect to elastically deformable models, discretized using FEM, can be found in [HS04, MG04, ITF06]. They proposed methods which compensate for the rotation of the elements. Liu *et al.* [LJWD08] also present a FEM-based GPU implementation. Their results show that the involved CG method dominates the total computation time.

Since FEM often involves a CG solver, considerable research was done on efficiently mapping the CG method and Sparse Matrix-Vector Multiplications (SPMV) on modern GPUs using CUDA, see [BG08, BCL07, VJ12] and the references therein. Other approaches for solving the resulting PDE use multigrid methods, see e.g. [GW06]. An efficient GPU implementation of a multigrid method, used for deformable models, was recently presented in [DGW11]. Although multigrid methods typically converge faster than CG methods, implementing them efficiently on a GPU is a much more elaborate process. For example, invoking an iterative solver such as CG, constitutes only one of the steps of a multigrid method, the others being smoothing, interpolation and restriction.

3 ELASTICITY THROUGH THE METHOD OF FINITE ELEMENTS

As common in computer graphics applications (see [MG04] and the references therein), we employ a linearized model based on *linear* elasticity theory [PH05]. Further, to solve the underlying PDE we use the Method of Finite Elements with *linear tetrahedral elements*.

3.1 Continuum elasticity

In continuum elasticity, the deformation of a body, *i.e.*, a continuous connected subset M of $\mathbb{R}^3$, is given by the *displacement* vector field $\boldsymbol{u}(\boldsymbol{x}) = [u(\boldsymbol{x}), v(\boldsymbol{x}), w(\boldsymbol{x})]^T$, where $\boldsymbol{x} = [x, y, z]^T$ is some point of the body at rest. Thus, every point $\boldsymbol{x}$ of the undeformed body corresponds to a point $\boldsymbol{x} + \boldsymbol{u}(\boldsymbol{x})$ of the deformed one.

The equilibrium equation of the deformation is usually written in terms of the *stress tensor*, $\boldsymbol{\sigma}$. However, since it cannot be measured directly, one uses Cauchy's *linear strain tensor*, $\boldsymbol{\varepsilon}$, and some material parameters to approximate the stress inside the body. Similar to Hooke's law for a 1D spring, in 3D one has

$$\boldsymbol{\sigma} = \boldsymbol{D} \cdot \boldsymbol{\varepsilon}, \tag{1}$$

for each point of the body, where $\boldsymbol{D} \in \mathbb{R}^{6\times 6}$ is the so-called *material stiffness matrix* representing material parameters. The elastic force $\boldsymbol{f}_e$ acting at a point of the body is given by

$$\boldsymbol{f}_e = \boldsymbol{K} \cdot \boldsymbol{u} = \left(\boldsymbol{P}^T \boldsymbol{D} \boldsymbol{P}\right) \cdot \boldsymbol{u}, \tag{2}$$

with $\boldsymbol{K} \in \mathbb{R}^{3\times 3}$, $\boldsymbol{f}_e$ and $\boldsymbol{u} \in \mathbb{R}^{3\times 1}$. $\boldsymbol{K}$ represents the local *stiffness matrix* and $\boldsymbol{P} \in \mathbb{R}^{6\times 3}$ is a matrix of partial derivative operators.

3.2 System dynamics

Having defined the elastical forces acting in a body, we now derive the equations of motion required to simulate the dynamic behaviour of the object. The coordinate vectors $\boldsymbol{x}$ are now functions of time, *i.e.* $\boldsymbol{x}(t)$, such that the equation of motion becomes

$$m\ddot{\boldsymbol{x}} + c\dot{\boldsymbol{x}} + \boldsymbol{f}_e = \boldsymbol{F}_{ext}, \tag{3}$$

where m is the mass of a body particle at position $\boldsymbol{x}$, c the damping coefficient, $\boldsymbol{f}_e$ the elastic force and $\boldsymbol{F}_{ext}$ the vector of external forces, i.e., the gravitational force. We approximate Eq. (3) using a *semi-implicit* method, *i.e.*,

$$m\frac{\left(\boldsymbol{v}^{i+1}-\boldsymbol{v}^{i}\right)}{\Delta t}+c\boldsymbol{v}^{i+1}+\boldsymbol{K}\cdot\boldsymbol{u}^{i+i}=\boldsymbol{F}^{i}_{ext}. \quad (4)$$

$$\boldsymbol{x}^{i+1}=\boldsymbol{x}^{i}+\Delta t\,\boldsymbol{v}^{i+1}, \quad (5)$$

with $\boldsymbol{u}^{i+1}=\Delta t\boldsymbol{v}^{i+1}+\boldsymbol{x}^{i}-\boldsymbol{x}^{0}$, which can be rearranged as

$$\left(m+\Delta tc+\Delta t^{2}\boldsymbol{K}\right)\cdot\boldsymbol{v}^{i+1}= m\boldsymbol{v}^{i}-\Delta t\left(\boldsymbol{K}\cdot\boldsymbol{x}^{i}-\boldsymbol{K}\cdot\boldsymbol{x}^{0}-\boldsymbol{F}^{i}_{ext}\right). \quad (6)$$

3.3 Discretization using FEM

Within FEM, the continuous displacement field $\boldsymbol{u}$ is replaced by a discrete set of displacement vectors $\tilde{\boldsymbol{u}}$ defined only at the nodes of the elements. Within each element e the displacement field is approximated by

$$\boldsymbol{u}=\boldsymbol{\Phi}_e\cdot\tilde{\boldsymbol{u}}, \quad (7)$$

where $\boldsymbol{\Phi}_e\in\mathbb{R}^{3\times 12}$ is the matrix containing the element *shape functions* and $\tilde{\boldsymbol{u}}=[u_1,v_1,w_1,\ldots,u_4,v_4,w_4]^T$ is the vector of the nodal displacement approximations. Next, Galerkin's method of weighted residuals is applied over the whole volume V, in which the *weighting* functions are equal to the shape functions. Each term in Eq. (6) is weighted and approximated as in Eq. (7), which results in

$$\int_V \boldsymbol{\Phi}^T\left(m+\Delta tc+\Delta t^{2}\boldsymbol{K}\right)\boldsymbol{\Phi}\cdot\tilde{\boldsymbol{v}}^{i+1}dV= \int_V m\boldsymbol{\Phi}^T\boldsymbol{\Phi}\tilde{\boldsymbol{v}}^{i}dV- \Delta t\int_V\boldsymbol{\Phi}^T\left(\boldsymbol{K}\boldsymbol{\Phi}\cdot\tilde{\boldsymbol{x}}^{i}-\boldsymbol{K}\boldsymbol{\Phi}\cdot\tilde{\boldsymbol{x}}^{0}-\boldsymbol{\Phi}\cdot\tilde{\boldsymbol{F}}^{i}_{ext}\right)dV, \quad (8)$$

with $\boldsymbol{\Phi}^T$ the weighting functions. The equation above is defined for each individual element and generates one matrix consisting of the local mass ($\boldsymbol{M}_e$), damping ($\boldsymbol{C}_e$) and element stiffness ($\boldsymbol{K}_e$) matrices. Additionally, a local force matrix ($\boldsymbol{F}_e$) is generated, representing the net external force applied to the object. These local matrices are given by

$$\begin{aligned}\boldsymbol{M}_e&=\rho_e\textstyle\int_V\boldsymbol{\Phi}_e^T\boldsymbol{\Phi}_e\,dV\\ \boldsymbol{C}_e&=c\textstyle\int_V\boldsymbol{\Phi}_e^T\boldsymbol{\Phi}_e\,dV\\ \boldsymbol{K}_e&=\textstyle\int_V\boldsymbol{\Phi}^T\boldsymbol{P}^T\boldsymbol{D}\boldsymbol{P}\boldsymbol{\Phi}\,dV\\ \boldsymbol{F}_e&=\textstyle\int_V\boldsymbol{\Phi}_e^T\boldsymbol{\Phi}_e\cdot\tilde{\boldsymbol{F}}_{ext}\,dV,\end{aligned} \quad (9)$$

with ρ_e the density of element e. See [PH05] for more details on computing these matrices.

Finally, the global matrix $\boldsymbol{K}\in\mathbb{R}^{3n\times 3n}$ (with n the number of mesh vertices) is 'assembled' from individual element matrices. This resulting system is then solved using the *Conjugate Gradient* method for the unknown velocity $\boldsymbol{v}^{i+1}$, which is then used to update the positions of the nodes, see Eq. (5). Eq. (5) shows a first order method for updating the positions which can be replaced by higher order methods as described in [ITF06].

Unfortunately, the above equations for simulating elastic deformation only work fine as long as the model does not undergo *large rotations*. This is because linearized elastic forces are used, which are only 'valid' close to the initial configuration. Therefore we use the so-called *Element-based Stiffness Warping* or *Corotational Strain* method [MG04, HS04] to compensate for the rotation of the elements. To extract the rotation part of the deformation, we use the polar decomposition method proposed in [Hig86]. The rotation-free element stiffness matrix $\boldsymbol{K}_{re}$ then becomes $\boldsymbol{K}_{re}=\boldsymbol{R}_e\boldsymbol{K}_e\boldsymbol{R}_e^{-1}$, with $\boldsymbol{R}_e\in\mathbb{R}^{12\times 12}$ the rotation matrix for element e. Note that this gives rise to an initial elastic force $\boldsymbol{f}_{e0}=\boldsymbol{R}_e\boldsymbol{K}_e\cdot\boldsymbol{x}_0$, which replaces the term $\boldsymbol{K}\boldsymbol{\Phi}\cdot\tilde{\boldsymbol{x}}^0$ in the right-hand-side of Eq. (8).

4 OVERVIEW OF THE ALGORITHM

Algorithm 1 gives an overview of the simulation of elastically deformable models as described in Section 3. First, a tetrahedralization of the polygonal mesh representing the surface of the object is computed, see Section 5.5. Each tetrahedron is considered as an element in FEM. Then, the initial stiffness-matrices of the elements are computed (line 3); these matrices do not change during the simulation and thus are precomputed. Additionally, as the shape functions are constant during the simulation, we can pre-calculate most matrices from Eq. (9), using $\boldsymbol{N}_1=\boldsymbol{\Phi}_e^T\boldsymbol{\Phi}_e$. This matrix is identical for all elements and is therefore only computed once.

Algorithm 1 Simulation algorithm.

1:	Compute $\boldsymbol{N}_1$	see Eq. (9)
2:	**foreach** element e	
3:	Compute $\boldsymbol{K}_e$	see Eq. (9)
4:	**loop of the simulation**	
5:	**foreach** element e	
6:	Compute volume v_e	
7:	Compute $\boldsymbol{R}_e$	see Section 3.3
8:	Compute $\boldsymbol{K}_{re}=\boldsymbol{R}_e\boldsymbol{K}_e\boldsymbol{R}_e^{-1}v_e$	
9:	Compute $\boldsymbol{M}_e=\rho_e\boldsymbol{N}_1v_e$	
10:	Compute $\boldsymbol{C}_e=c\boldsymbol{N}_1v_e$	
11:	Compute $\boldsymbol{f}_{e0}=\boldsymbol{R}_e\boldsymbol{K}_e\cdot\boldsymbol{x}_0v_e$	
12:	Compute $\boldsymbol{F}_e=\boldsymbol{N}_1\cdot\tilde{\boldsymbol{F}}_{ext}v_e$	see Eq. (9)
13:	Compute $\boldsymbol{F}_{te}=\boldsymbol{M}_e\cdot\boldsymbol{v}^i-\Delta t\left(\boldsymbol{f}_{e0}-\boldsymbol{K}_{re}\cdot\boldsymbol{x}^i-\boldsymbol{F}_e\right)$	
14:	Compute $\boldsymbol{K}_{te}=\boldsymbol{M}_e+\Delta t\boldsymbol{C}_e+\Delta t^2\boldsymbol{K}_{re}$	
15:	Assemble global $\boldsymbol{K}$ and $\boldsymbol{F}$ using $\boldsymbol{K}_{te}$ and $\boldsymbol{F}_{te}$ of elements	
16:	Solve $\boldsymbol{K}\cdot\boldsymbol{v}^{i+1}=\boldsymbol{F}$ for $\boldsymbol{v}^{i+1}$	
17:	Update $\boldsymbol{x}^{i+1}=\boldsymbol{x}^i+\Delta t\boldsymbol{v}^{i+1}$	see Section 3.2

After all local matrices have been computed and stored (line 14), the global matrix is assembled

(line 15). The resulting linear system of equations is solved for velocities (line 16), which are then used to advance the position vectors (line 17).

5 GPU MAPPING USING CUDA

In this section we describe our GPU mapping of the simulation on NVida GeForce GPUs using CUDA [NVI]. First we shall give details about implementing the rotation extraction through polar decomposition. Then, we describe the computation of the local stiffness matrices which are used to assemble the global (sparse) stiffness matrix (matrix $\boldsymbol{K}$ in Algorithm 1). The resulting system of linear equations is solved using a Jacobi-Preconditioned CG Method.

5.1 Rotation extraction

As mentioned in subsection 3.3 we have to estimate the rotation of each element in order to calculate displacements properly. Finding the rotational part of the deformation matrix is done using a Polar Decomposition as described in [MG04, HS04, Hig86]. Although a large number of matrix inversions is involved, this can be done efficiently because small 4×4 matrices are used. Since each matrix contains 16 elements, we chose to map the computations of 16 such matrices to a single CUDA thread-block with 256 threads.

For computing the inverse of a 4×4 matrix we perform a *co-factor expansion*. Each thread within a thread-block computes one co-factor of the assigned matrix. Since the computation of a co-factor requires almost all values of the matrix, memory accesses have to be optimized. In order to prevent for possible bank-conflicts during the computation of the co-factors, each matrix is stored in one memory bank of shared memory. Accordingly, the shared-memory segment (of size 16×16 locations) is regarded as a matrix stored in row-major order, where each column represents a 4×4 local matrix. Therefore, each column (local matrix) maps exactly to one memory-bank. Since a large number of rotation matrices are computed in parallel, a large performance boost is obtained.

5.2 Local stiffness matrices

Solving a specific problem using FEM starts with describing the problem locally per element. Since a typical problem consists of a large number of elements, the computations involved per element can be easily parallelized. Further, since the matrices used to construct $\boldsymbol{K}_e$ are in $\mathbb{R}^{12\times12}$, we map the computation of each individual local element stiffness matrix to a thread-block containing 12×12 threads. The inner loop in Algorithm 1 is implemented using one or two CUDA kernels, depending on the architecture version. Instead of creating kernels for each individual matrix operation, we combine a number of them into one larger kernel. Since data from global memory can be reused multiple times, less global memory transactions are required, which improves the overall performance.

5.3 Solving the linear system

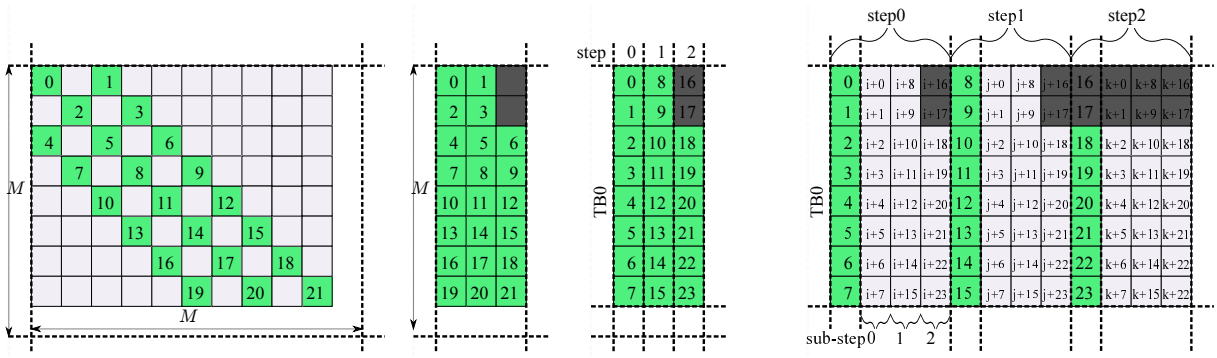
Given the local element matrices and load vectors, the global stiffness matrix of the system is assembled. Next, the system has to be solved for the unknown velocity $\boldsymbol{v}_{i+i}$. The (Jacobi-Preconditioned) CG method performs a large number of sparse matrix-vector multiplications and other vector-vector operations. Therefore, solving a large linear system efficiently, requires a fast and efficient implementation of sparse matrix-vector multiplications, which is highly-dependent on the layout used for storing the sparse matrix. Since three unknown values (components of the velocity vector) are associated to each mesh vertex, a block with 3×3 elements in the global matrix corresponds to each edge of the tetrahedral mesh. Therefore, a *Block-Compressed Sparse Row* (BCSR) format is very well suited for storing the global matrix, and thus improving the speed of the CG method.

Furthermore, since the vertex degree of internal nodes is constant in a regular tetrahedralization (see sect 5.5), the variation of the number of elements per row in the global matrix is minimal. Therefore, we use the optimized BCSR format from [VJ12]. This method efficiently stores a large sparse-matrix in BCSR format and reorders the blocks in memory to improve the efficiency of the memory transactions. This fact is very important since the main bottleneck of the CG method is the memory throughput. In [VJ12], through extensive experiments, it is shown that their optimized BCSR layout outperforms other storage formats for efficient matrix-vector multiplication on the GPU.

5.4 Global matrix update

Each local matrix represents a set of equations for each individual tetrahedron. To obtain the global system of equations, each component of each local matrix is added to the corresponding location of the global matrix. The location is obtained using the indices of the vertices for that specific element. Since the structure of the underlying mesh does not change during the simulation, also the structure of the global matrix remains unchanged. Therefore we assemble the global matrix only once and updates its values every time step. In this section, we propose an extension of [VJ12] which allows us to efficiently update a sparse matrix stored in the BCSR format.

For updating the global matrix, two approaches are possible. Within the first approach (*scatter*), all values of a local matrix are added to their corresponding values in the global matrix. When the local matrices are processed on the GPU, many of them are processed in parallel. Therefore, multiple threads *can* update the same



(a) Block layout of a sparse-matrix: Each green block stores $N \times N$ values and its position within the block-row. Numbers represent memory locations. Each gray block contains zero-values and is not explicitly stored. M represents the dimension of the matrix.

(b) BCSR layout: Each block-row is compressed; an additional array with indices to the first block in a block row is necessary (not shown here).

(c) Blocks of consecutive block rows are mapped to a thread block ($TB0$). Blocks mapped to the same thread block are reordered so that blocks processed in the *same step* are continuous in memory. Padding is required (gray blocks).

(d) Updating matrix blocks (green), requires the associated local values. The indices of these values are stored in *index blocks* (gray), in the same order as the matrix blocks. Within each sub-step, a set of continuous index-blocks are loaded and used to fetch the corresponding values from the local matrices. The dark-gray blocks are used for padding and contain -1's. i, j, k represent (starting) offsets in memory.

Figure 2: Updating the sparse matrix: the initial sparse matrix is created, stored and processed, (a), (b) and (c). Updating the matrix is done by collecting the corresponding values from the local matrices, (d).

value in the global matrix at the same time, which results in *race conditions*. In order to prevent race conditions from appearing, access to the values of the global matrix would have to be serialized.

The second approach is to *gather* per element in the global matrix, the corresponding values from the local matrices. To do so, the indices of all associated local values are stored per element in the global matrix. Each index represents the position of the local value in an array A, which stores the values of all local matrices. Given these indices per global element value, the local values are looked-up and used to update the corresponding value in the global matrix.

Within the optimized BCSR implementation of [VJ12], the global sparse-matrix is divided in $N \times N$-sized blocks, Fig. 2(a). Next, block rows are compressed and sorted by length, Fig. 2(b). Finally, a number of consecutive block rows are grouped and mapped to a CUDA thread block. Within each group of block rows, the blocks are reordered in memory, such that accessing these blocks is performed as optimal as possible. Accessing the blocks (for e.g. a multiplication) is done as follows. First, all threads of a thread-block ($TB0$) are used to access the blocks mapped to it in the first step (step 0), see Fig. 2(c). Each thread computes an index pointing to these blocks. Next, blocks $0-7$ are loaded from the global memory. Note that these are the same blocks appearing in the first column of Fig. 2(b). For the next step, each thread increases its current index, such that the next set of blocks $(8-15)$ can be loaded (step 1). Note that all block rows must have the same length, and therefore, empty blocks must be padded (blocks 16 and 17).

To actually update the data blocks of the global matrix, we use a *gather* approach. Accordingly, $N \times N$-sized *index blocks* are used for each matrix block, see Fig. 2(d). Since the matrix blocks have a specific ordering, the same ordering is used for the index-blocks. For each step, a number of *sub-steps* is performed. Within each sub-step a set of index-blocks is loaded from memory, given a start offset (i, j or k in Fig. 2(d)). Then, for each index-block, its $N \times N$ values (indices) are used to fetch the corresponding $N \times N$ data values from local matrices, stored in global memory. Please note that the $N \times N$ data values fetched using one $N \times N$ index-block, do not come, in general, from the same local matrices. To accumulate the local contributions, an array (stored in shared memory) is used. If an index has value -1, no update is performed. For the next sub-step, the indices pointing to the index blocks are increased. Therefore, per step, the number of index blocks for each processed matrix block must be equal, which requires padding with '-1' index blocks. The advantage of this approach is that loading the indices and writing the updated values always result in an optimal throughput. Loading the actual local-element values is in general not optimal.

5.5 Tetrahedralization and rendering

The quality of the tetrahedral mesh is essential for efficiently simulating a deforming elastic object represented by a polygonal mesh. We have experimented with tetrahedralizations in which the surface mesh forms the outer boundary of the tetrahedral mesh. Since the tri-

angles of the surface mesh can have a high variety in size, the generated tetrahedralization also contains tetrahedral elements with a high variation in size and configuration. This can have a negative effect on the quality of the tetrahedralization. Therefore, we chose to create a tetrahedral mesh, using equi-sized elements, which however, may result in a rather rough approximation of the original surface mesh. We tackle this problem by coupling the input polygonal mesh to the (deforming) tetrahedral mesh, as follows.

First, a regular 3D grid of N^3 voxels is created, in which each voxel containing a part of the surface is marked as important; typical values for N are $32, 64$ or 128. Next, a regular tetrahedralization of the grid is created using equi-sized tetrahedral elements, and each element containing at least one important vertex of the grid, is stored. Further, the inner volume of the object is tetrahedralized using the same equi-sized tetrahedral elements. Next, in order to reduce the amount of elements, those elements belonging to the inner volume are merged together into fewer larger ones. This reduces the total amount of elements and thus the total computation time. Note however that this approach is most useful with models which have large internal volumes, similar to the bunny in Figure 5. Finally, the original surface mesh is coupled with the tetrahedral one similar to [MG04]: each vertex in the original surface mesh is mapped to exactly one tetrahedron, and its barycentric coordinates in that tetrahedron are stored along with the vertex coordinates.

When the new positions of the tetrahedra are computed, the surface mesh is also updated. To compute new positions of the deformed surface mesh, for each vertex of the input mesh, the positions of the four vertices of the corresponding tetrahedron are looked-up and interpolated using the barycentric coordinates of the original vertex.

6 RESULTS

All experiments have been performed on a machine equipped with an Intel Q6600 quad-core processor and a GeForce GTX 570 with 1.2 Gb of memory.

Figure 3 shows the performances obtained for computing the local element matrices (*Matrix*), the rotation matrices (*Rotation*), solving the resulting linear system (*CG*), performing a single SpMV (*SpMV*), and the total performance (*Total*) as a function of the number of elements. *Steps/sec* is the corresponding number of simulation steps performed per second. Similarly, Figure 4 shows the computation time per simulation time-step. For each model, we have used the following material parameters: Young's modulus of elasticity, $E = 5 \times 10^5 N/m^2$; Poisson's ratio, $\mu = 0.2$; density, $\rho = 1000 KG/m^3$. Furthermore, the time step of the simulation $\Delta t = 0.001$ and the volume of each initial element $v_e = 1.65 \times 10^{-6} m^3$. Each model used in

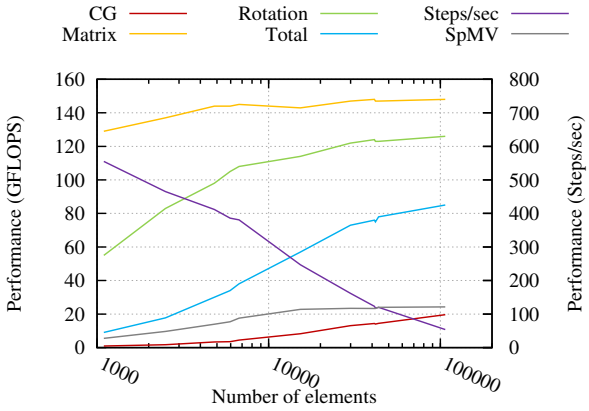


Figure 3: Performance results with different numbers of elements. *CG* represents the performance of the CG solver, *Matrix* – the performance for computing the local element matrices, *Rotation* – the performance of the rotation extraction procedure, *SpMV* – the performance of the SpMV operation; *Total* represents the overall performance. *Steps/sec* represents the number of simulation steps per second. The global-matrix update was performed with an effective throughput of 50 GB/sec.

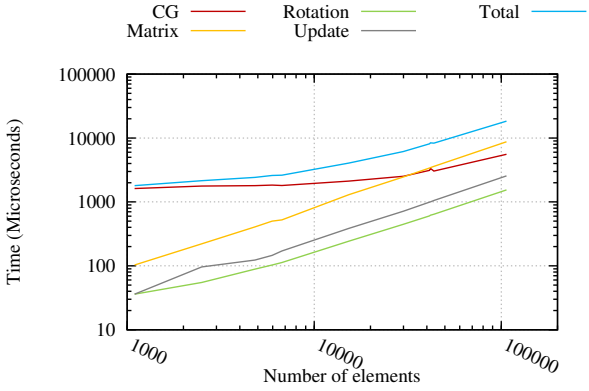


Figure 4: Timing results with different numbers of elements, per time step. *CG* represents the time of the CG solver, *Matrix* – the time for computing the local element matrices, *Rotation* – the time of the rotation extraction procedure; *Total* represents the total elapsed time per time-step.

this paper is scaled such that each dimension is at most 66 *cm* and is tetrahedralized as described in Section 5.5. With these settings, the CG solver found a solution for each model in 5 to 18 iterations. In order to obtain a generic performance picture, we have fixed the number of iterations to 18, which resulted in the performances from Fig. 3.

Within Figure 3 a number of interesting patterns can be seen. First, the performance for computing the local element matrices reaches its maximum very soon. Since each matrix is mapped to exactly one thread-

block, a large amount of thread-blocks is created, resulting in a 'constant' performance. Second, the performance figures for computing the rotation matrices show a larger variation. Since 16 rotation matrices are processed by one thread-block, a significantly smaller amount of thread-blocks is used. Finally, the performance of the CG method seems to be low compared to the other operations. The CG method operates on a global sparse-matrix and performs a large number of sparse-matrix vector multiplications (SPMVs) and vector-vector operations, for which the performances are mainly bound by the memory throughput. However, the CG performances from Figure 3 agree with those from [VJ12], given the dimensions of the problem.

The measured, effective throughput for updating the global matrix was about 50 GB/sec, in all cases with more than $5k$ elements. Since this operation transfers a large amount of data, the memory bus is saturated very soon, resulting in a good throughput. However, since not all transactions can be coalesced, the maximum throughput is not reached. This operation is very similar to an SPMV with 1×1 blocks, but now for a matrix containing $d\times$ more elements, with d the degree of internal nodes in the model. This observation shows that the measured throughput is close to the expected one, according to the results in [VJ12].

As expected, the total performance increases with the number of elements. This shows that the computational resources are used efficiently for larger models. The number of elements, for which the maximum performance is reached, depends on the actual GPU mapping of the computations. For example, the CG solver does not reach its maximum performance for $100k$ elements, while the computation of the local element matrices reaches its peak at $5k$ elements. Due to this, one can expect better performances for the CG method when larger models are used. Furthermore, for models having less than $30k$ elements, the total computation is dominated by the time spent by the CG solver. For larger models, more time is spent on computing the local matrices, see Figure 4.

The measured overall performance is based on the total time needed per simulation step, which includes all operations performed, except the rendering of the model. Figure 3 also shows the number of simulation steps performed per second, given the number of elements; these numbers are based on the total computation time. Accordingly, even for large models, interactive frame rates can be reached. A rough comparison of the obtained performance and frame rate with other state-of-the-art multigrid GPU implementations [DGW11] shows that, even if in theory the CG method converges slower than multigrid, comparable results *can* be obtained for similar models. We assume that memory transactions in our method are more efficient, despite of transferring more data. However, more research is required to get a full understanding of the differences between both methods performed on modern GPUs, with respect to performance figures. Finally, Figures 1, 5, 6, 7, 8 and 9 show example results from our simulations.

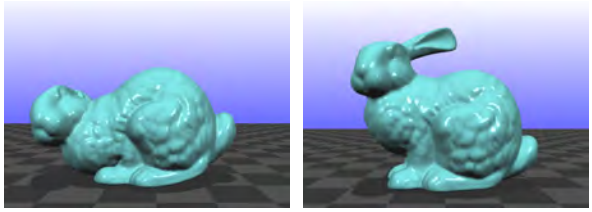


Figure 5: Material properties and collision handling. *Left*: flexible material ($E = 5 \times 10^4$). *Right*: stiffer material ($E = 5 \times 10^5$). Simulation rate: 120 frames per second.

Figure 6: Left: stretching and deforming a model using external forces. Right: deformation after releasing external forces. Simulation rate: 118 frames per second.

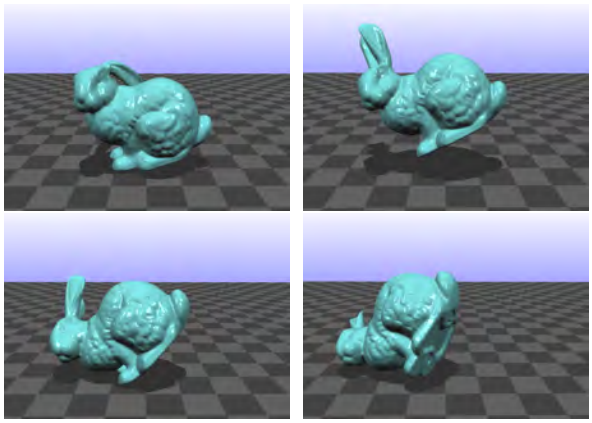


Figure 7: Bunny bouncing on the floor. Simulation rate: 120 frames per second.

7 CONCLUSIONS

We have presented an efficient method for simulating elastically deformable models for graphics applications, accelerated on modern GPUs using CUDA. Our method relies on a fast Conjugate Gradient solver and an efficient mapping of the SPMV operation on modern GPUs [VJ12]. Since the topology of the underlying grid

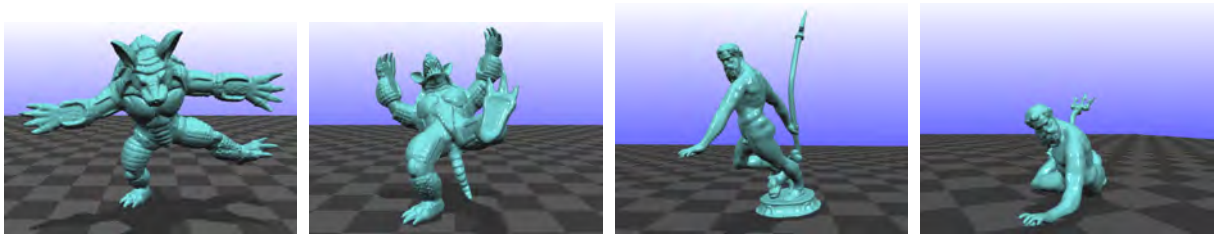


Figure 9: Other simulation results. Simulation rate: 160 frames per second.

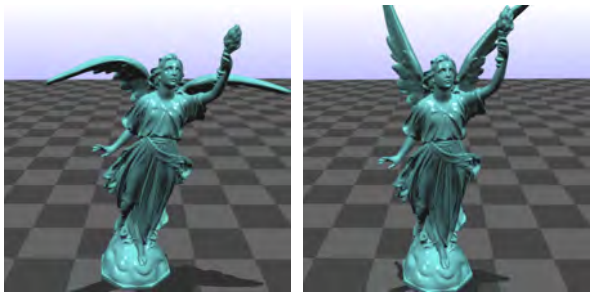


Figure 8: Left: applying external forces on the wings. Right: after releasing the external forces. Simulation rate: 116 frames per second.

does not change during the simulation, data structures are reused for higher efficiency. To further improve the performance, we proposed a scheme which allows to efficiently update the sparse matrix, during the simulation.

In future work we will investigate the performance of this method when multiple GPUs are used. Furthermore, we will investigate the performance difference between traditional CG methods and multigrid methods performed on modern GPUs. Also, we plan to enhance the simulation to allow for plastic behaviour as well as brittle and fracture of stiff materials.

REFERENCES

[BCL07] BUATOIS L., CAUMON G., LÉVY B.: Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU. In *High Perf. Comp. Conf. (HPCC)* (2007). 2

[BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHRÖDER P.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *Proc. SIGGRAPH'03* (2003). 2

[BG08] BELL N., GARLAND M.: *Efficient Sparse Matrix-Vector Multiplication on CUDA*. Tech. Rep. NVR-2008-004, NVidia, 2008. 2

[DGW11] DICK C., GEORGII J., WESTERMANN R.: A real-time multigrid finite hexahedra method for elasticity simulation using CUDA. *Simulation Modelling Practice and Theory 19*, 2 (2011), 801–816. 2, 7

[GM97] GIBSON S., MIRTICH B.: *A Survey of Deformable Modeling in Computer Graphics*. Tech. Rep. TR-97-19, MERL, Cambridge, MA, 1997. 1

[GSMY*07] GÖDDEKE D., STRZODKA R., MOHD-YUSOF J., MCCORMICK P., BUIJSSEN S. H., GRAJEWSKI M., TUREK S.: Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing 33*, 10–11 (2007), 685–699. 2

[GST05] GÖDDEKE D., STRZODKA R., TUREK S.: Accelerating double precision FEM simulations with GPUs. In *Proc. ASIM 2005 - 18th Symp. on Simul. Technique* (2005). 2

[GST07] GÖDDEKE D., STRZODKA R., TUREK S.: Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *Int. Journal of Parallel, Emergent and Distributed Systems 22*, 4 (2007), 221–256. 2

[GW06] GEORGII J., WESTERMANN R.: A multigrid framework for real-time simulation of deformable bodies. *Computers & Graphics 30*, 3 (2006), 408–415. 2

[Hig86] HIGHAM N. J.: Computing the polar decomposition – with applications. *SIAM Journal of Scientific and Statistical Computing 7* (1986), 1160–1174. 3, 4

[HS04] HAUTH M., STRASSER W.: Corotational simulation of deformable solids. In *WSCG* (2004), pp. 137–144. 2, 3, 4

[ITF06] IRVING G., TERAN J., FEDKIW R.: Tetrahedral and hexahedral invertible finite elements. *Graph. Models 68*, 2 (2006), 66–89. 2, 3

[JP99] JAMES D. L., PAI D. K.: ArtDefo: accurate real time deformable objects. In *Proc. SIGGRAPH'99* (1999), pp. 65–72.

1

[KW03] KRÜGER J., WESTERMANN R.: Linear algebra operators for gpu implementation of numerical algorithms. In *Proc. SIGGRAPH'03* (2003), pp. 908–916. 2

[LJWD08] LIU Y., JIAO S., WU W., DE S.: Gpu accelerated fast fem deformation simulation. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on* (30 2008-dec. 3 2008), pp. 606 –609. 2

[MG04] MÜLLER M., GROSS M.: Interactive virtual materials. In *Proc. Graphics Interface 2004* (2004), pp. 239–246. 2, 3, 4, 6

[NMK*06] NEALEN A., MÜLLER M., KEISER R., BOXERMANN E., CARLSON M.: Physically based deformable models in computer graphics. *Computer Graphics Forum 25* (2006), 809–836. 1

[NVI] NVIDIA CORPORATION: *Compute Unified Device Architecture programming guide*. Available at `http://developer.nvidia.com/cuda`. 4

[PH05] PEPPER D. W., HEINRICH J. C.: *The Finite Element Method: Basic Concepts and Applications*. Taylor and Francis, 2005. 1, 2, 3

[RS01] RUMPF M., STRZODKA R.: Using graphics cards for quantized FEM computations. In *Proc. IASTED Vis., Imaging and Image Proc.* (2001), pp. 193–202. 2

[TBHF03] TERAN J., BLEMKER S., HING V. N. T., FEDKIW R.: Finite volume methods for the simulation of skeletal muscle. In *In SIGGRAPH/Eurographics symposium on Computer Animation* (2003), pp. 68–74. 1

[TPBF87] TERZOPOULOS D., PLATT J., BARR A., FLEISCHER K.: Elastically deformable models. In *Proc. SIGGRAPH'87* (1987), pp. 205–214. 1

[VJ12] VERSCHOOR M., JALBA A. C.: Analysis and performance estimation of the conjugate gradient method on multiple gpus. *Parallel Computing* (2012). (in press). 1, 2, 4, 5, 7

VISUALIZATION OF FLOW FIELDS IN THE WEB PLATFORM

Mauricio Aristizabal
Universidad EAFIT
[1]CAD/CAM/CAE Laboratory
Carrera 49 # 7 Sur - 50
Colombia (050022), Medellin, Antioquia
maristi7@eafit.edu.co

John Congote[1]
Vicomtech Research Center
Mikeletegi Pasealekua, 57
Parque tecnológico
Spain (20009), Donostia - San Sebastian, Guipuzcoa
jcongote@vicomtech.org

Alvaro Segura
Vicomtech Research Center
Mikeletegi Pasealekua, 57
Parque tecnológico
Spain (20009), Donostia - San Sebastian, Guipuzcoa
asegura@vicomtech.org

Aior Moreno
Vicomtech Research Center
Mikeletegi Pasealekua, 57
Parque tecnológico
Spain (20009), Donostia - San Sebastian, Guipuzcoa
amoreno@vicomtech.org

Harbil Arregui
Vicomtech Research Center
Mikeletegi Pasealekua, 57
Parque tecnológico
Spain (20009), Donostia - San Sebastian, Guipuzcoa
harregui@vicomtech.org

Oscar E. Ruiz
Universidad EAFIT
CAD/CAM/CAE Laboratory
Carrera 49 # 7 Sur - 50
Colombia (050022), Medellin, Antioquia
oruiz@eafit.edu.co

ABSTRACT

Visualization of vector fields plays an important role in research activities nowadays. Web applications allow a fast, multi-platform and multi-device access to data, which results in the need of optimized applications to be implemented in both high and low-performance devices. The computation of trajectories usually repeats calculations due to the fact that several points might lie over the same trajectory. This paper presents a new methodology to calculate point trajectories over a highly-dense and uniformly-distributed grid of points in which the trajectories are forced to lie over the points in the grid. Its advantages rely on a highly parallel computing implementation and in the reduction of the computational effort to calculate the stream paths since unnecessary calculations are avoided by reusing data through iterations. As case study, the visualization of oceanic streams in the web platform is presented and analyzed, using WebGL as the parallel computing architecture and the rendering engine.

Keywords

Streamlines, Trajectory, Hierarchical Integration, Flow Visualization, WebGL.

1 INTRODUCTION

Vector field visualization plays an important role in the automotive and aero-spatial industries, maritime transport, engineering activities and others. It allows the detection of particularities of the field such as vortexes or eddies in flow fields, but also permits exploring the entire field behavior, determining flow paths.

In particular, ocean flow visualization is important in maritime navigation and climate prediction, since the movement of sea water masses produces variations in air temperature and wind currents. Therefore, flow visualization becomes determinant to represent the ocean's behavior.

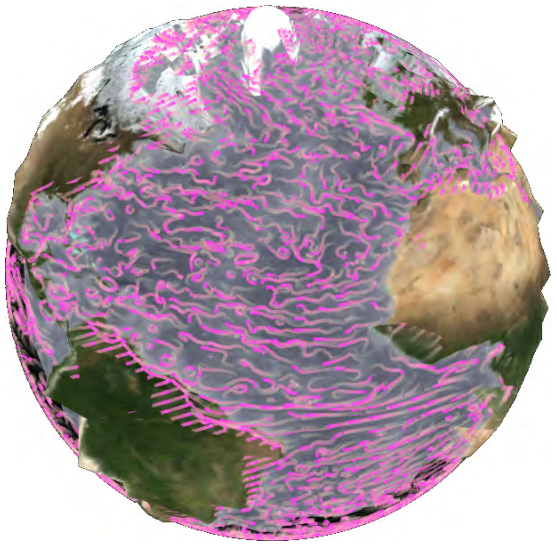


Figure 1: flow visualization of Atlantic ocean currents in WebGL. Hierarchical integration was used to reduce the total number of iterations required to calculate the paths.

With the growth of a great diversity of devices, development of multi-platform applications has become a common goal for developers. The Web is de-facto a universal platform to unify the development and execution of applications. However, challenges arise since applications must be optimized in order to be useful as well as on high as on low-performance devices.

The development of parallel computing hardware for all the different devices is increasing and the development of applications and computer-based procedures are taking in advance this capability. The contribution of this paper is a new methodology to calculate point trajectories of a highly dense grid of points over n-dimensional vector fields, in which the trajectories are forced to pass over the grid points (Figure 1). This allows to implement a hierarchical integration procedure ([HSW11]), which takes advance of previously calculated data in order to avoid repetitive and unnecessary calculations, and reduces the complexity of the algorithm from linear to logarithmic. The procedure is suitable to be implemented over highly parallel computing architectures due to independent calculations and the number of computations to be performed. We employ WebGL as the parallel computing engine to calculate the iterations, using the inherently parallel rendering procedure, and images are used to store the data through the iterations.

Different from other procedures, in which the calculation of the trajectories is performed for each point in particular, our methodology allows to merge its calculation for all the points in which the field is discretized. Therefore, the number of unnecessary computations is critically reduced.

This paper is organized as follows: Section 2 presents the related work. Section 3 exposes the methodology in which the contribution of this work is explained. Section 4 presents a case of study in oceanic currents and finally section 5 concludes the article.

2 LITERATURE REVIEW

2.1 Flow Visualization

A great amount of methodologies to visualize vector fields (flow fields) has been developed among the last decades. Geometric-based approaches draw icons on the screen whose characteristics represent the behavior of the flow (as velocity magnitude, vorticity, etc). Examples of these methodologies are arrow grids ([KH91]), streamlines ([KM92]) and streaklines ([Lan94]). However, as these are discrete approaches, the placement of each object is critical to detect the flow's anomalies (such as vortexes or eddies), and therefore, data preprocessing is needed to perform an illustrative flow visualization. An up-to-date survey on geometric-based approaches is presented by [MLP10].

However, in terms of calculating those trajectories for determined points in the field, the procedures usually compute for each point the integrals, and, as a result, the procedures are computationally expensive for highly dense data sets.

On the other hand, texture-based approaches represent both a more dense and a more accurate visualization, which can easily deal with the flow's feature representation as a dense and semi-continuous (instead of sparse and discrete) flow visualization is produced. A deep survey in the topic on texture-based flow visualization techniques is presented by [LHD04].

An animated flow visualization technique in which a noise image is bended out by the vector field, and then blended with a number of background images is presented by [VW02]. Then, in [VW03] the images are mapped to a curved surface, in which the transformed image visualizes the superficial flow.

Line Integral Convolution (LIC), presented by [CL93], is a widely implemented texture-based flow visualization procedure. It convolves the associated texture-pixels (texels) with some noise field (usually a white noise image) over the trajectory of each texel in some vector field. This methodology has been extended to represent animated ([FC95]), 3D ([LMI04]) and time varying ([LM05, LMI04]) flow fields.

An acceleration scheme for integration-based flow visualization techniques is presented by [HSW11]. The optimization relies on the fact that the integral curves (such as LIC) are hierarchically constructed using previously calculated data, and, therefore, avoid unnecessary calculations. As a result, the computational effort is reduced, compared to serial integration techniques, from $O(N)$ to $O(\log N)$, where N refers to the number of steps to calculate the integrals. Its implementation is performed on Compute Unified Device Architecture (CUDA), which allows a parallel computing scheme performed in the Graphics Processing Unit (GPU), and therefore the computation time is critically reduced. However, it requires, additionally to the graphic Application Programming Interface (API), the CUDA API in order to reuse data, and hence, execute the procedure.

2.2 WebGL literature review

The Khronos Group released the WebGL 1.0 Specification in 2011. It is a JavaScript binding of OpenGL ES 2.0 API and allows a direct access to GPU graphical parallel computation from a web-page. Calls to the API are relatively simple and its implementation does not require the installation of external plug-ins, allowing an easy deployment of multi-platform and multi-device applications. However, only images can be transfered between rendering procedures using framebuffer objects (FBOs).

Several WebGL implementations of different applications have been done such as volume rendering, presented by [CSK11] or visualization of biological data,

presented by [CADB10]. A methodology to implement LIC flow visualization with hierarchical integration, using only WebGL, was presented by [ACS12], in which FBOs are used to transfer data between different rendering procedures, and therefore allowing to take in advance the parallel computing capabilities of the rendering hardware, in order to perform the different calculations. However, for the best of our knowledge, no other implementation that regards to streamline flow visualization on WebGL has been found in the literature or in the Web.

2.3 Conclusion of the Literature Review

WebGL implementations allow to perform applications for heterogeneous architectures in a wide range of devices from low-capacity smart phones to high-performance workstations, without any external requirement of plug-ins or applets. As a result, optimized applications must be developed. In response to that, this work optimizes the calculation of point trajectories in n-dimensional vector fields over highly dense set of points, forcing the trajectories to lie over the points in the set. As a consequence, previously calculated data can be reused using hierarchical integration, avoiding unnecessary calculations and reducing the complexity of the algorithm.

3 METHODOLOGY

The problem that we address is stated as: given a set of points and a vector field that exists for all of these points, the goal is to find the finite trajectory that each point will reproduce for a certain period of time.

Normal calculation of point trajectories in n-dimensional vector fields, requires to perform numerical integration for each particular point in order to reproduce the paths. In the case of a dense set of points, the procedures suffer from unnecessary step calculations of the integrals, since several points in the field might lie over the same trajectory of others. Hence, some portions of the paths might be shared. Figure 2 illustrates this situation.

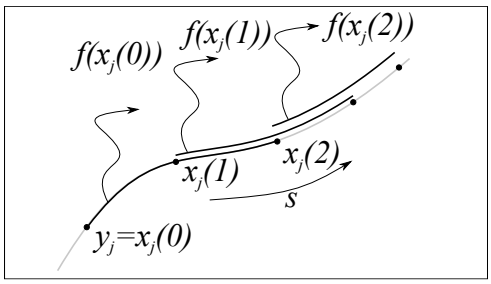


Figure 2: Trajectory overlapping in several point paths.

In order to avoid repeated computations, we propose a methodology to calculate trajectories of highly dense grid of points, in which the paths are forced to lie over the points in the grid, i.e., the paths are generated as a Piecewise Linear (PL) topological connection between a set of points that approximates the trajectory. With this, hierarchical integration [HSW11] is employed to iteratively compute the paths and reuse data through the iterations.

3.1 Hierarchical Integration

Since line integration over n-dimensional vector fields suffers from repeated calculations, hierarchical integration [HSW11] only calculates the necessary steps and then iteratively grows the integrals reusing the data. This reduces the computational complexity of the algorithm from $O(N)$, using serial integration, to $O(\log N)$. The procedure is summarized as follows.

For an arbitrary point in the field $y \in Y$ with $Y \subseteq \mathbb{R}^n$, let us define $f : Y \rightarrow \mathbb{R}^m$, as an arbitrary line integral bounded by its trajectory c_y. Consider its discrete approximation as described in equation 1.

$$f(y) = \int_{c_y} w(x(s))ds \approx \sum_{i=1}^{t} w(x(i*\Delta s))\Delta s \qquad (1)$$

where t is the maximum number of steps required to reproduce c_y with Δs the step size. $x(0) = y$ is the starting point of the trajectory to be evaluated and w is the function to be integrated. The integration procedure is performed for all points $y \in Y$ in parallel.

We assume that $\Delta s = 1$, $\forall y \in Y$ and therefore $f(y) \approx \sum_{i=1}^{t} w(x(i))$. The algorithm starts with the calculation of the first integration step for all the points in the field. Namely,

$$f_0(y) = w(x(1)) \qquad (2)$$

It is required to store the last evaluated point $x(1)$ over the growing trajectory and the partial value of the integral for all the points y in order to reuse them in the following steps to build the integral. With this, the next action is to update the value of the integral, using the sum of the previously calculated step at y and the step evaluated at its end point ($x(1)$). Namely,

$$f_1(y) = f_0(x(0)) + f_0(x(1)) \qquad (3)$$

In this case, the end point of $f_1(x(0))$ is $x(2)$ as the calculation evaluates $f_0(x(1))$. Therefore, the next iteration must evaluate f_1 at $x(0)$ and $x(2)$ in order to grow the integral. In general, the k'th iteration of the procedure is calculated as follows:

$$f_k(y) = f_{k-1}(x(0)) + f_{k-1}(x(end)) \qquad (4)$$

It is important to remark that each iteration of this procedure evaluates two times the integration steps evaluated in the previous iteration. As a result, the total number of integration steps t is a power of two,

and the hierarchical iterations required to achieve this evaluations is reduced by a logarithmic scale, i.e., $k = \log_2 t$. Also notice that the evaluation of the vector field is performed only once, in the calculation of the first step, which avoids unnecessary evaluations of the vector field, which are computationally demanding for complex vector fields. Figure 3 illustrates the procedure up to four hierarchical steps.

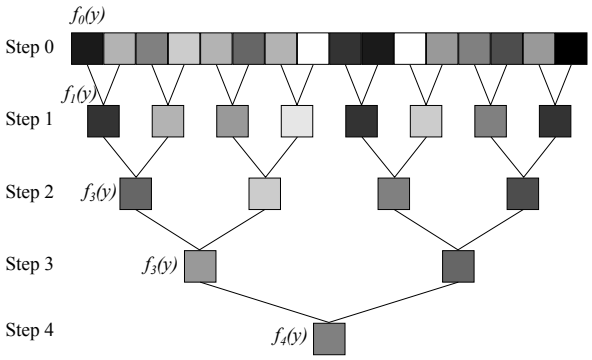


Figure 3: Exponential growth of hierarchical integration methodology. At step 3, the procedure evaluates 8 serial integration steps, meanwhile at step 4 it evaluates 16 serial integration steps.

3.2 Stream Path Calculation

In order to perform the visualization of a vector field using trajectory paths, lets assume a homogeneously distributed set of points

$$Y = \{y,\ z \in \mathbb{R}^n | y - z = \Delta y, \\ \Delta y \text{ is constant } \forall z \text{ adjacent to } y\} \quad (5)$$

and a n-dimensional vector field $F : \mathbb{R}^n \to \mathbb{R}^n$. The goal is to calculate for each point $y \in Y$, the PL approximation of the trajectory that the point will describe according to F, defined by the topological connection of a particular set of points $A_y \subset Y$. Figure 4 illustrates the approximation.

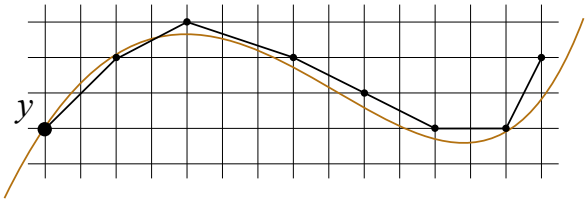


Figure 4: PL approximation of the trajectory by the procedure.

The trajectory c_y of an arbitrary point y in the field is defined as

$$c_y = x_y(s) = \int_l F(x_y(s))ds \quad (6)$$

where l represents a determined length of integration.

Using hierarchical integration, for each point in the field the first step of the PL trajectory is calculated, this is, the corresponding end point of the first step of the integral is computed using a local approximation of the point in the set of points.

$$x_y(0) = y \quad (7)$$

$$y' = x_y(0) + \gamma F(x_y(0)) \quad (8)$$

$$x_y(1) = \arg\min_{x_y}(Y - y') \quad (9)$$

where y' is the first iteration result of the Euler integration procedure, γ is a transformation parameter to adjust the step given by the vector field to the local separation of the set of points and $x_y(1)$ is defined as the closest point in Y that approximates y'. The value of $x_y(1)$ is then associated (and stored) to y. The set A_y contains the reference to the points of the trajectory that describes y, and therefore for equation 8, A_y is defined as:

$$A_y^0 = \{x_y(1)\} \quad (10)$$

Similarly to the hierarchical integration procedure, the next steps are performed to join the calculated steps in order to grow the trajectories. Therefore, for each point y, its computed trajectory is joint with its last point's trajectory, this is, for the step in equation 8.

$$A_y^1 = A_y^0 \cup A_{x_y(1)}^0 = \{x_y(1), x_y(2)\} \quad (11)$$

Note that each iteration of the procedure will increase the number of points in the trajectory by a power of two. Therefore, the growth of the paths is exponential. In general, the k'th iteration is calculated as

$$A_y^k = A_y^{k-1} \cup A_{x_y(2^k)}^{k-1} = \{x_y(1), x_y(2), \ldots, x_y(2^{(k+1)})\} \quad (12)$$

The accuracy of the procedure is strongly determined by the discretization (density) of Y, since it is directly related to the step size in the integration procedure, i.e., the approximation of the first step end-point is determinant. In order to increase the accuracy of the procedure, the computation of the first step can be calculated with e.g., a 4th order Runge Kutta numerical integration, however, it might significantly increase the computation time of the procedure if the computation time of the vector field function is relevantly high.

3.3 Time-Varying Data

In terms of unsteady flow fields, i.e., time-varying vector fields, the generation of the trajectories might seem difficult. In that case, as proposed in [HSW11], time is considered another dimension of the vector field.

Therefore, the set of points is formed with the position coordinates of the points and discretized time steps, producing an $n+1$-dimensional grid of points.

It is also determinant for the accuracy of the procedure that the density of the discretization set is high, in order to increase the precision of the approximated trajectories.

3.4 Animation

Dynamic scenes are demanding in most of the visualization procedures. We consider in this section two kinds of dynamic scenes. A first kind of procedures refers to when the vector field is steady, i.e., it remains constant through the time. In this case, the goal is to visualize the motion of the particle all across the field.

Since the paths for all the points in the field are calculated, the representation of the particle's trajectory through the frames is simple. Consider a point y and its approximated trajectory given by the set of points A_y. Notice, as described in section 3.2, that the first point of the set $A_y[1]$, i.e., $x_y(1)$, represents the next point in which y will lie in a determined period of time. As a result, at a posterior frame, the displayed trajectory should be $A_{x_y(1)}$.

The second type of procedure refers when vector field is varying with the time. Complementary to the animation stated before, a second kind of dynamic scene is comprised since it is also important to visualize the changes that a trajectory suffers in the time. In the case of time varying data, as in the steady case, all the points have an associated trajectory. In order to animate the change of one trajectory, from one frame to another, the trajectory that will be represented refers to the one of the point with the same point coordinate, but the next time coordinate. i.e., $A_{y,t+\Delta t}$.

4 CASE STUDY

In this section the visualization of 2D oceanic currents using the proposed methodology is performed. The implementation has been done in WebGL, so the methodology's parallel computing capabilities are fully used. WebGL offers the possibility to use the rendering procedure to calculate images (textures) through Framebuffer Objects, and then use those rendered textures as input images for other rendering procedures. As a consequence, for this implementation we associate the pixels of an image to the points in the field, and therefore, the rendering procedure is used to compute the different hierarchical iterations, which are stored in the color values of the pixels. Finally, the trajectories are hierarchically constructed. The implementation was performed on an Intel Core2Quad 2.33 GHz with 4 GB of RAM and with a nVidia GeForce 480.

4.1 Implementation

For a $w \times h$ grid of points (w and h being its width and height respectively in number of elements), images of size $w \times h$ in pixels are used, in which a particular pixel (i, j) is associated with the point (i, j) in the grid. Since for each particular pixel, a four component vector is associated, i.e., a vector of red, green, blue and alpha values, each value can be associated as a particular position of another pixel. This is, if the value of a pixel is r, then its associated pixel coordinates are given by

$$i = r \bmod w \tag{13}$$

$$j = \frac{r-i}{w} \tag{14}$$

where mod represents the remainder of the division of r by w. As a result, if for each hierarchical integration, only the last point of the trajectory is to be stored, then one image can store four hierarchical iterations.

For the zero'th hierarchical iteration, and the image I to store its calculation, the value of a pixel (i, j) is given by

$$i_0 = i + kF_i(i, j) \tag{15}$$

$$j_0 = j + kF_j(i, j) \tag{16}$$

$$\tag{17}$$

where the parameter '0' refers to the hierarchical step 0, k represents the scaling factor of the vector field, and $F_i(i, j)$ represents the component of the vector field over the direction of i, evaluated at the point (i, j). The vector field used in this case study is shown in figures 5(a) for the direction of i and 5(b) for the direction of j.

In general, the k'th step is calculated as follows,

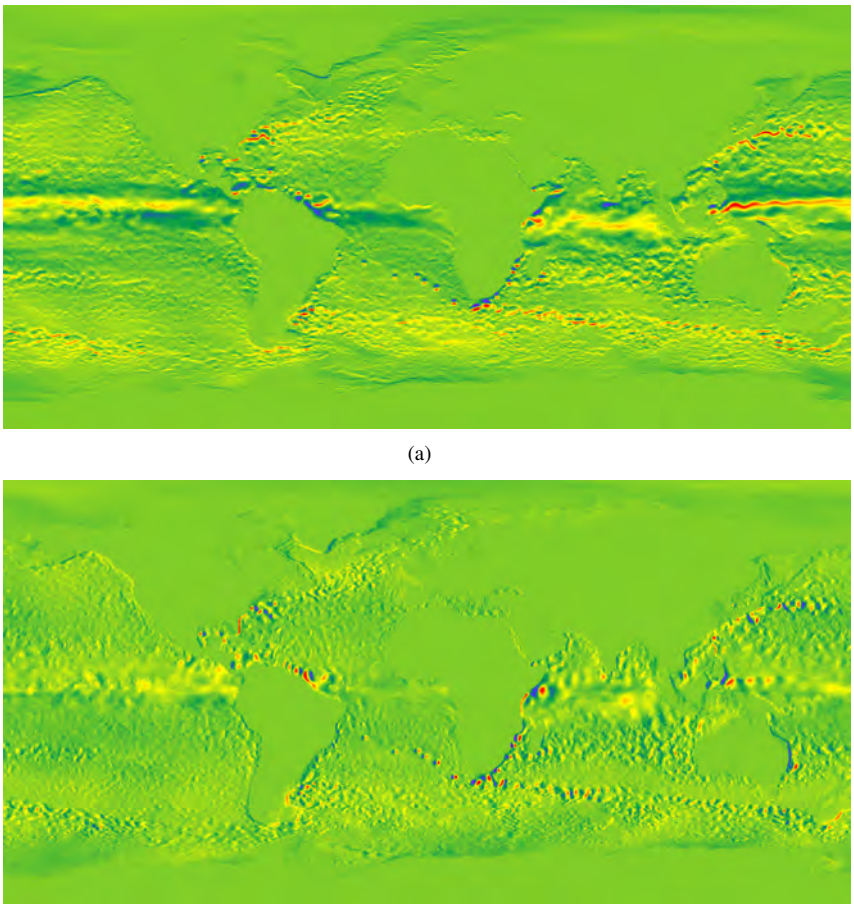
$$i_{next} = I(i, j, k-1) \bmod w \tag{18}$$

$$j_{next} = \frac{I(i, j, k-1) - i_{next}}{w} \tag{19}$$

$$I(i, j, k) = I(i_{next}, j_{next}, k-1) \tag{20}$$

In the case that k is greater than four, then more images are used to store the values of the hierarchical iterations.

With that, all the desired steps are stored in the necessary images. In order to build the trajectories from those images, a generic line, formed by k^2 points, is required. Each point in the trajectory needs to have an associated value, that refers to its order in the trajectory, i.e., the first point in the trajectory has an index 0, the second point the value 1 and so on. With this index associated to each point of the trajectory of the point y, the position of each point is calculated as described in Algorithm 1, where HL is the hierarchical level that needs to be evaluated and the function $evalHL()$ returns the new position of the point y, for a particular hierarchical level.



(a)

(b)

Figure 5: Oceanic currents information. Color-scale of the oceanic currents magnitude in the (a) longitudinal and (b) latitudinal directions, of the 1^{st} January of 2010. Images generated using data from the NASA's ECCO2 project.

Require: $index$ Index referring to the element position into the line.
y Position of the initial point of the trajectory.

Ensure: y_{end} Position of the i'th element of the line in the space.

Finished = $false$
while not Finished **do**
  $HL = \text{floor}(\log_2(index))$
  $index = index - 2^{HL}$
  $y = \text{evalHL}(HL, y)$
  if $index == 0$ **then**
    $y_{end} = y$
    Finished = $true$

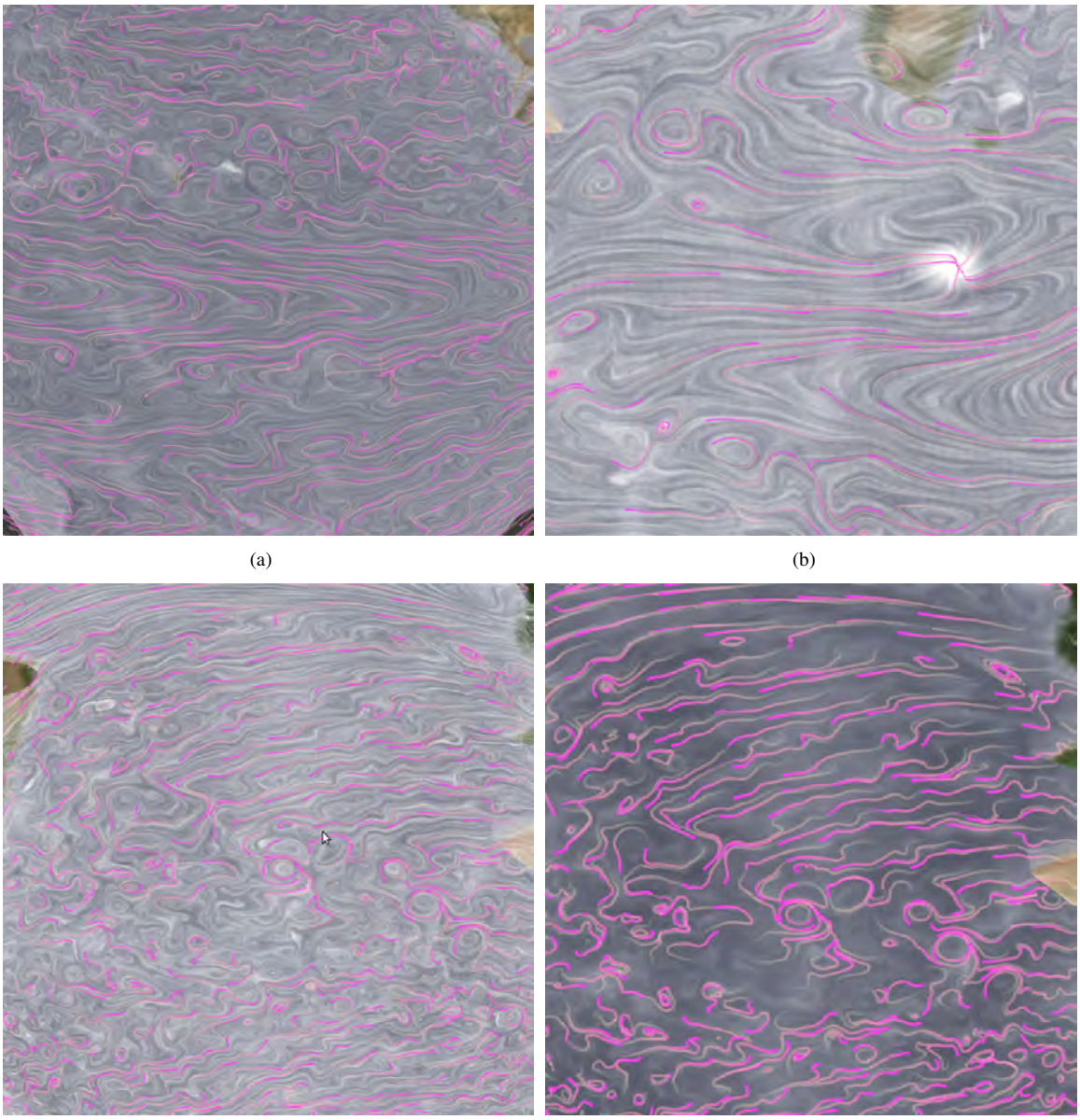
Algorithm 1: Procedure to reconstruct the streamlines using the already calculated hierarchical levels.

4.2 Results

A general grid of 2048×2048 points is used as the world's discretization. The vector field information was acquired from the NASA's ECCO2 project (see figure 5), in which high-resolution data is available. Only six hierarchical levels, i.e., $2^6 = 64$ points in the trajectory are used for each point in the field, as a result only 2 images are required to calculate the trajectories.

The time needed to compute all the hierarchical levels (from 0 to 6) was 3 ms. The trajectory computation was performed to 10000 equally-distributed points all over the field, which means that 64000 points need to be transformed by the trajectory computation. The computation time of those trajectories was 670 ms (Final results are shown in Figure 6).

In order to compare the visualization using the proposed methodology, the LIC visualization of the vector field using the methodology proposed in [ACS12] was inserted below the stream line visualization. It shows that for this level of discretization (2048×2048), the visualization is correct and accurate. However, sparse data might produce inaccurate results.



(a) (b)

(c) (d)

Figure 6: Different view points of the final visualization of the oceanic currents using hierarchically calculated streamlines. Six hierarchical steps were used to achieve this visualization. The LIC visualization of the flow is used below the streamlines in order to visually compare the different methods and to enhance the visualization

5 CONCLUSIONS AND FUTURE WORK

This article presents a novel methodology to calculate points trajectories in highly dense point sets, in which the trajectories are formed as piecewise-linear connections of the points in the set. This allows to merge the calculation of the different trajectories and use iteratively the data to avoid repeated and unnecessary calculations, hence, accelerating the process. The procedure is suitable to be implemented in parallel architectures such as WebGl, OpenCL or CUDA, since the calculations of the integrals for one point are independent from the calculation of the other points.

As a result, thanks to the use of hierarchical integration, the procedure reduces the computational complexity of the calculation of the trajectories from linear to logarithmic. The methodology deals with n-dimensional and time-varying data and animated visualization can be easily achieved due to the fact that the trajectories are calculated for all the points in the set.

Since the procedure performs an approximation of the trajectories using a piecewise-linear connection of the points in the set, the accuracy of the algorithm is strongly influenced by the discretization distance between the points, because this distance determines a lower bound in the integration step to be used.

Ongoing research focuses on the adaptation of this methodology to require less computational effort (processing and memory effort), so that extremely low-performance devices such as smart-phones and tablets might be able to perform an accurate and complete flow visualization using streamlines. Related future work includes the adjustment of the grid point positions along the iterations and the increase in the accuracy of the calculated trajectories.

ACKNOWLEDGEMENTS

This work was partially supported by the Basque Government's ETORTEK Project (ITSASEUSII) research program and CAD/CAM/CAE Laboratory at EAFIT University and the Colombian Council for Science and Technology COLCIENCIAS. The vector field information was acquired form NASA's ECCO2 project in *http://ecco2.jpl.nasa.gov/*.

6 REFERENCES

[ACS12] Mauricio Aristizabal, John Congote, Alvaro Segura, Aitor Moreno, Harbil Arriegui, and O. Ruiz. Hardware-accelerated web visualization of vector fields. case study in oceanic currents. In Robert S. Laramee Paul Richard, Martin Kraus and José Braz, editors, *IVAPP-2012. International Conference on Computer Vision Theory and Applications*, pages 759–763, Rome, Italy, February 2012. INSTICC, SciTePress.

[CADB10] M. Callieri, R.M. Andrei, M. Di Benedetto, M. Zoppè, and R. Scopigno. Visualization methods for molecular studies on the web platform. In *Proceedings of the 15th International Conference on Web 3D Technology*, pages 117–126. ACM, 2010.

[CL93] B. Cabral and L.C. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 263–270. ACM, 1993.

[CSK11] J. Congote, A. Segura, L. Kabongo, A. Moreno, J. Posada, and O. Ruiz. Interactive visualization of volumetric data with webgl in real-time. In *Proceedings of the 16th International Conference on 3D Web Technology*, pages 137–146. ACM, 2011.

[FC95] L.K. Forssell and S.D. Cohen. Using line integral convolution for flow visualization: Curvilinear grids, variable-speed animation, and unsteady flows. *Visualization and Computer Graphics, IEEE Transactions on*, 1(2):133–141, 1995.

[HSW11] M. Hlawatsch, F. Sadlo, and D. Weiskopf. Hierarchical line integration. *Visualization and Computer Graphics, IEEE Transactions on*, (99):1–1, 2011.

[KH91] R.V. Klassen and S.J. Harrington. Shadowed hedgehogs: A technique for visualizing 2d slices of 3d vector fields. In *Proceedings of the 2nd conference on Visualization'91*, pages 148–153. IEEE Computer Society Press, 1991.

[KM92] D.N. Kenwright and G.D. Mallinson. A 3-d streamline tracking algorithm using dual stream functions. In *Proceedings of the 3rd conference on Visualization'92*, pages 62–68. IEEE Computer Society Press, 1992.

[Lan94] D.A. Lane. Ufat: a particle tracer for time-dependent flow fields. In *Proceedings of the conference on Visualization'94*, pages 257–264. IEEE Computer Society Press, 1994.

[LHD04] R.S. Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F.H. Post, and D. Weiskopf. The state of the art in flow visualization: Dense and texture-based techniques. In *Computer Graphics Forum*, volume 23, pages 203–221. Wiley Online Library, 2004.

[LM05] Z. Liu and R.J. Moorhead. Accelerated unsteady flow line integral convolution. *IEEE Transactions on Visualization and Computer Graphics*, pages 113–125, 2005.

[LMI04] Z. Liu and R.J. Moorhead II. Visualizing time-varying three-dimensional flow fields using accelerated uflic. In *The 11th International Symposium on Flow Visualization*, pages 9–12. Citeseer, 2004.

[MLP10] T. McLoughlin, R.S. Laramee, R. Peikert, F.H. Post, and M. Chen. Over two decades of integration-based, geometric flow visualization. In *Computer Graphics Forum*, volume 29, pages 1807–1829. Wiley Online Library, 2010.

[VW02] J.J. Van Wijk. Image based flow visualization. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 745–754. ACM, 2002.

[VW03] J.J. Van Wijk. Image based flow visualization for curved surfaces. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 17. IEEE Computer Society, 2003.

Hybrid SURF-Golay Marker Detection Method for Augmented Reality Applications

David Prochazka, Ondrej Popelka, Tomas Koubek, Jaromir Landa, Jan Kolomaznik
Mendel University in Brno
Zemedelska 1
61300, Brno, Czech Republic
david.prochazka@mendelu.cz

ABSTRACT

Augmented reality is a visualization technique widely used in many applications including different design tools. These tools are frequently based on tracking artificial objects such as square markers. The markers allow users to add a 3D model into the scene and adjust its position and orientation. Nevertheless, there are significant problems with marker occlusions caused by users or objects within the scene. The occlusions usually cause a disappearance of the 3D model. Such behavior has substantial negative impact on the application usability. In this article we present a hybrid marker detection approach. With this approach, markers are detected using the well-known SURF method. This method is able to recognize complex natural objects and deal with partial occlusions. Further, we overcome the problem of distinguishing similar markers by using the Golay error correction code patterns. The described approach represents a robust method that is able to identify even significantly occluded markers, differentiate similar markers, and it works in a constant time regardless of the amount of used markers.

Keywords

Augmented reality, augmented prototyping, SURF, Golay error correction code.

1. INTRODUCTION

The augmented reality (AR) research has been running for almost two decades. Nevertheless, it is possible to find just a few applications for common users. There are several principal reasons. One of the key problems is the inability to deal with occlusions of markers that are used for scene augmentation. During the work with an AR application, a marker is frequently obstructed by different solid objects, e.g. **users´** hands. Inability to identify such **a** partially occluded marker leads to frequent disappearances of a visualized 3D model. Despite the obvious importance, this problem is unsolved even in many well-known AR toolkits (e.g. *ARToolKitPlus*).

The presented approach is implemented in the AR application *AuRel* that is focused on an augmented prototyping process**.** The application is developed in cooperation with an automotive company. It allows a car designer to extend a physical car model by selected virtual objects (3D models of car spare parts (see Fig. 1)). The usage of AR for industrial design is mentioned in many papers, e.g. in [FAM*02], [BKF*00] and [VSP*03].

Figure 1: 3D model of a spoiler inserted onto a rear car hood

Although there is a substantial amount of existing augmented reality frameworks (*ARToolkit*, *ARTag*, *Studierstube*, etc.), *OpenCV* library has been used for the implementation [Lag11]. Principal reasons being active *OpenCV* development, cross-platform

deployment, 64bit systems support, a wide range of implemented computer vision algorithms and the amount of documentation (books, tutorials, etc.) [PK11].

There are briefly summarized current methods used for recognition of possible markers in the section 2. Two approaches focused on identification of geometric features are compared with the advanced technique usually used for natural object detection. Further, the section 3 outlines our method that is composed of SURF marker detection and Golay error correction code identification. Finally, the section 4 presents our results and concentrates on the ability to deal with occlusions.

2. MARKER RECOGNITION METHODS

The process of marker recognition is usually divided in two parts: *marker detection* and *marker identification*. The former involves recognition of video frame regions that may represent markers. The latter concentrates on verifying the identity of the markers. The marker identity defines which 3D model will be displayed to the user.

2.1 Marker Detection Approaches

The registration process of all further described methods is influenced by many negative factors, e.g. low image resolution, camera distortion (caused by lens), various light conditions or marker occlusions. The methods endeavor to compensate most of these factors. For the purpose of the article, the methods are distinguished into three general groups according to their basic principles**.** In detail the description of object recognition methods can be found e.g. in [Sze11].

2.1.1 Morphology-based marker detection

These methods are based on recognition of shapes in preprocessed images. An approach described in [HNL96] uses a system of *concentric contrast circles* (CCC). The marker is composed of a black circle around a white middle or vice versa. The detection process starts with image thresholding and noise removal. Further, connected components are found, and their centers are determined. The results are two sets of centers: centers of white connected components and centers of black connected components. CCC marker position is given by the cross section of black and white centers**.**

An example of another approach is implemented in the frequently used *ARToolKit* [KB99]. In this case, square markers with black borders and black-and-white inner pictures are detected. A camera image is thresholded and connected components contours are found. Further, quadrangles are selected from the contours set. These quadrangles represent potential markers [KTB*03].

The obvious limitation of these methods is their inability to deal with occlusions. Such occlusion causes a change in the image morphology. Therefore, the required shape cannot be detected.

2.1.2 Edge-based marker detection

These methods are more flexible with regard to the marker occlusions than the image morphology-based methods. One solution that is based on this principle is the *ARTag* system [Fia05]. Although the *ARTag* markers are similar to the *ARToolKit* markers (significant is a thick black border), the implemented detection method is completely different. The *ARTag* method is based on detection of *edgels* (edge pixels) of an object. Further, a set of lines is constructed from the found *edgels*. It is not necessary to detect all line *edgels*; therefore, the edge could be partially occluded. Finally, four corresponding lines represent edges of a potential marker.

The same detection principle is used also in *StudierStube* project [Hir08] and many others.

2.1.3 Feature-based marker detection

These methods are based on key points detection. The key points are various regions of interest: edges, corners, blobs. To identify whether a given key point really represents a part of a marker, it is necessary to match it with a key point in a marker template. This matching process requires that both key points to be matched are described by gradient changes in their neighborhood. The process of key point

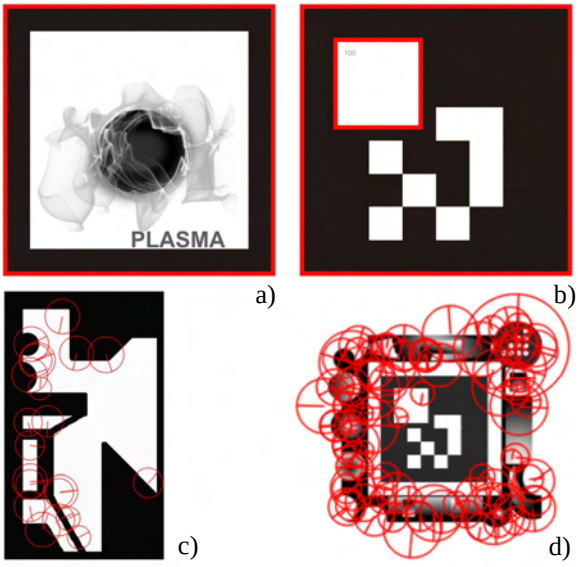


Figure 2: Markers used with different detection methods. Detected features are highlighted with a red color. From left: a) Match template, b) Golay error correction code, c) SURF and d) S-G detection

neighborhood description is usually called feature extraction. The output of this process is a set of feature descriptors vectors. The feature descriptors are later compared and their distance is computed [Low04]. This enables to match points of interest between a template and a camera image.

There are several approaches for feature-based detection. Widely used are e.g. SIFT [CHT*09] and SURF [BTG06]. A thorough comparison of selected methods is described in [TM08]. The SURF (Speeded Up Robust Features) algorithm has a good ratio between detection capabilities and performance. The SURF algorithm application is composed of three steps: detection of key points (points of interest), feature extraction and key point matching.

The detection of image key points that are used for the image description is based on gradient changes in the grayscale version of the image. Each key point is identified by position and radius that specifies the size of the key point neighborhood. Then the process of feature extraction is performed.

During this process, each key point neighborhood is described using 64-dimensional or 128-dimensional vector that describes the gradient changes of each key point neighborhood. The descriptors are produced both for a template and a camera image, so that the corresponding key points are identified.

The SURF main advantage is the scale and rotation invariance [BTG06]. This allows the SURF to work even with low resolution images or small objects. Another advantage is that the algorithm compares only the points; therefore, the object can be partially occluded. Although the SURF method is usually used for natural object identification (see e.g. [BCP*08]), it can be used also for marker detection as described in our method outlined in section 3.

2.2 Marker Identification Approaches

Morphological and edge-based detection methods are commonly used with following marker identification approaches: template matching and decoding of various binary codes.

Match template identification is based on computation of a pixel value correlation between a potential marker and a list of templates. In case the correlation fulfills a given threshold, the marker is identified. Obviously, the method has a linear time complexity. It is necessary to compute correlations with all templates until the required one is found or all templates are tested. Moreover, it is difficult to choose a threshold that allows to distinguish a large amount of markers [Bru09]. Therefore, methods based on different binary codes are frequently used to compensate this problem. One of the possible codes is the Golay error correction code.

A marker based on the Golay error correction code (ECC) can be composed of a large white square in the top left corner and e.g. 24 black or white squares that encode a number. The large square provides information about the marker orientation (see Fig. 2-b).

In the first step, a Golay ECC decoder for such a marker detects the position of the large white square. Further, it divides the code area into 24 blocks and calculates an average pixel value in all segments. Finally, the average value is thresholded to zero or one **a**nd the binary code is reconstructed. Possible implementation of the code reconstruction is outlined in [MZ06].

A significant advantage of this approach is that the binary code is reconstructed in a constant time. Another important advantage is the ability to correct errors caused by occlusions or an image corruption. Finally, the amount of distinguishable markers is limited just by the binary code length.

A feature-based method, such as the SURF is, is capable of both marker detection and marker identification. Therefore, it is not usually used with an identification method. As mentioned above, the method relies on searching for distinctive key points in a camera image that are then matched against image template descriptors. This process has linear time complexity because all template descriptors must be tested until the required one is found.

2.3 Summary of the Marker Recognition

In general, there are three approaches for marker recognition. The first one is based on image morphology. Detection can be fast and precise. However, it cannot deal with marker occlusions. The edge-based methods can detect partially occluded markers. Nevertheless, this ability is limited**.** Larger occlusions of the edges are problematic. Both detection methods can be accompanied by a binary code identification method that is able to work in a constant time and reliably distinguish a substantial amount of markers.

Feature-based approaches are able to detect and identify even substantially occluded markers. However, they work in a linear time. This complexity is usually problematic for real-time applications with larger amounts of markers. Even more, feature-based methods have problems with distinguishing of similar markers [SZG*09].

3. S-G HYBRID RECOGNITION METHOD

The proposed identification method combines the positive properties of two previously mentioned methods. We take advantage of robustness of the SURF feature-based object identification and

combine it with high reliability and effectiveness of the Golay error correction code detection, hence the name *S-G hybrid detection method*.

3.1 Marker design

As described in section 2.1.3, the SURF algorithm is suitable especially for natural objects identification. However, many applications use this method to identify only a single object in an image. This is marker may appear in a scene.

The most problematic part of the SURF marker identification is the matching of corresponding marker key points in both images. The key points similarity is determined by gradient changes in the key points neighborhoods (these are represented by feature descriptors). If the image contains areas with similar gradient changes, then such areas will be identified as the same or similar key points.

Therefore, it is important to design markers so that the key points identified in them have distinctive gradient changes. Furthermore, these key points must be distinguishable both from the scene image and from other markers.

We use artificial markers very distinctive from the scene objects. Acceptable results are obtained using complex asymmetric markers composed of arbitrary geometric shapes (see Fig. 2–c). These markers are easily detected because they contain a substantial amount of features which can be tracked. The development of such marker, however, requires a lot of manual work. Even with a thorough testing it seems that only a very low number (approx. 3) of these markers could be reliably distinguished in an image.

Therefore, to ensure the correct marker identification we propose a hybrid detection method – *S-G Detection* – in which we combine the SURF algorithm with the Golay error correction code. In this case, the marker template is divided into two parts: the marker border and marker content. These two parts of a template may be combined independently.

Marker content is composed solely of a Golay code image. Only the marker content is used for marker identification. This has the advantage of very high identification reliability and allows to distinguish large number of markers – see section 2.2.

Marker border is composed of different geometric shapes selected so that they are distinctive from real scene objects. However, the border is no longer used for identification of the markers. This is possible because each marker border may be combined with any number of Golay codes to identify the marker. This combination solves the problem of distinguishing between marker templates while maintaining great robustness against template occlusion (see Fig. 4). Different marker borders may be used in the application. However, it is not necessary. We use the same border for all markers.

3.2 Marker Detection

As been described in the previous section, we use the SURF method to identify key points only in the marker border (see Fig. 2–d). This border is the same for all markers. A strong advantage of this approach is that the time complexity of the whole algorithm is not a function of a number of templates (see section 2.1). Therefore, we can use a high number of markers without a performance hit. This is an important usability feature.

A common approach [Lag11] in matching the template and video frame points of interest is: find the best matches of key points using a defined metric, filter out false positives (invalid matches), repeat filtering until a sufficient number of adequately reliable points are obtained.

Errors in matched points may occur when a template key point is matched to an unrelated video frame key point because it happens to have similar neighborhood. Another source of errors occurs when a video frame contains two or more markers and template points are matched to correct points but on different marker borders (two or more physical markers).

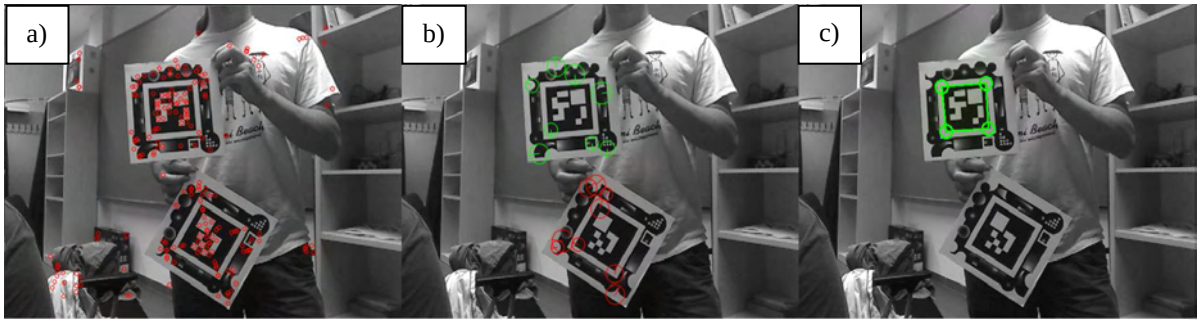


Figure 3: S-G hybrid method. From left: a) key points are detected and filtered b) angle filter is applied so that the key points on both markers are distinguished c) marker specified in application configuration is detected.

For many applications, it is enough to identify if the template is present in the image, other applications require approximate template positions. Our application requires the exact position (translation and rotation) of the marker so that the virtual object may be inserted to the real scene.

The first step of marker matching feature extractor is to discover key points in the processed image. Then a descriptor vector for each key point is found using a feature extractor. These vectors are matched by computation of Euclidean distance between each pair of points. Moreover, we use symmetric matching filter for the key points.

First, template key points are matched against video frame image and the best matches are selected. Then the frame key points are matched against template key points, and best matches are selected. The intersection of these two sets is a set of matched points [Lag11].

Further, we filter the set of key points by application of an angle filter. The idea behind the angle filter is to take advantage of the information stored in a SURF key point itself. Each SURF key point contains an angle value, which defines the direction of the most significant gradient descent in the neighborhood of the key point. In our application, we use artificial markers; therefore we search for a set of predefined objects. This means that relative differences in rotation of the matched key point must be similar for all matched key points. That is – if the template has two key points and their rotation is 45° and 70°, then the two key points matched in the frame must have the rotation difference approximately 25°. Due to perspective deformations, the differences can be computed only approximately. An example of this filtering is shown in Fig. 3 – each set of differently colored points maintains the same relative rotation differences between points (in other words the same rotation difference between a template and a video frame).

A difficult part of the angle filtering algorithm is defining initial conditions. This is caused by the fact that until the marker is identified, its key points, their rotations and order are all unknown. To overcome this problem, the angle filter algorithm is implemented by marshaling all possible rotations into overlapping intervals of a defined width (rotation difference tolerance – RT). Each interval overlaps half of neighboring intervals so that there are no artificial boundaries. Key points in each interval are then processed individually as if it was a standalone key point set. This introduces a performance hit as another loop iterating over sets of key point has to be processed. Fortunately this is upper bounded – maximum number of iterations is 360 / (RT · 2). This upper bound is hardly reached because only sets containing at least four points need to be processed. A minimum of four points is required for a correct positioning of a 3D model which will be added to the image later in the process. The angle filter algorithm may be described by the following pseudo-code:

```
FOR each matched_point
  difference =
    matched_point_template->angle -
    matched_point_frame->angle;
  div = difference / RT
  angles[div * RT]->add(matched_point)
  angles[(div + 1) * RT]
       ->add(matched_point)
END FOR
FOR each angle
  find homography
  identify Golay marker
  IF marker identified THEN
       display 3D object
END FOR
```

3.3 Marker Identification

For each set of points detected by the angle filter, we compute homography matrix so that the Golay code can be identified. By applying the homography transformation to the camera image we compensate the perspective deformation. This image transformed

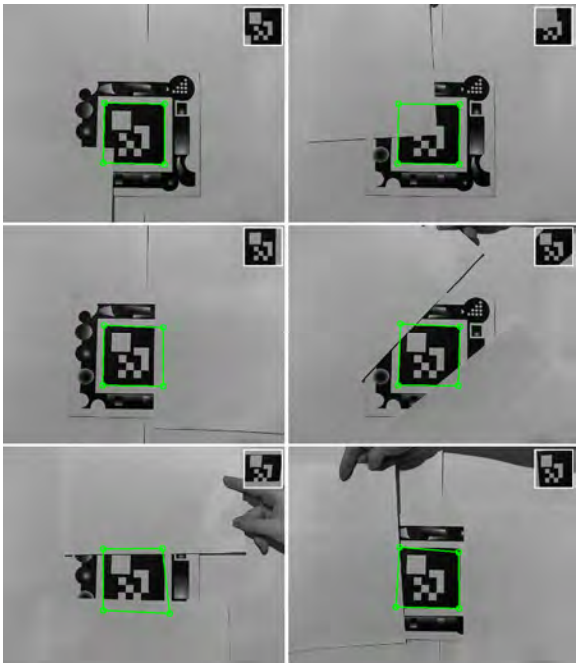


Figure4: Examples of S-G method capability of occluded marker identification from a close distance.

to the camera plane is cropped and processed by the Golay code detector.

If a Golay code is found, it means that the marker is identified. This identification introduces important feedback for the SURF marker detection. Given the reliability of the Golay detector, false positives are almost impossible. In other words, if the code is identified, we can be sure it is one of searched markers. It also means that the homography was computed correctly. This is also important because we can use the points to compute projection matrix. Reliable projection matrix is important for correct 3D models positioning.

In section 2.2 that describes the Golay codes is stated that the Golay code rotation is determined by the position of the large white square in the top left corner. Since the S-G detection method is focused on robustness against marker occlusions, it is undesirable to have parts of the marker with greater importance. In the S-G method, the rotation of the marker is determined solely by the position of key points. This part of the Golay code is therefore unused.

4. COMPARISON

The S-G hybrid method was tested against two other solutions: *ARToolKitPlus* (http://handheldar.icg.tugraz.at/artoolkitplus.php) and *ALVAR Toolkit* (www.vtt.fi/multimedia/alvar.html). All tests were made in a laboratory under artificial light. We used markers with 14 cm long edge for testing. The solutions were tested from three aspects:

- Distance – minimum, maximum and maximum distance without visible jitter.
- Angles – marker was placed at different distances from the camera and rotated around *x* and *y* axis (the *z* axis was not tested because all solutions are capable of 360 degrees rotation).
- Occlusion – occlusion was tested with stationary marker and camera.

Compared to the other two solutions, S-G has a smaller maximum distance where it is capable to identify a marker. The S-G method is able to detect a marker placed at a distance 2 m from the camera. The *ARToolKitPlus* and *ALVAR* have maximal distance at approx. 5 m.

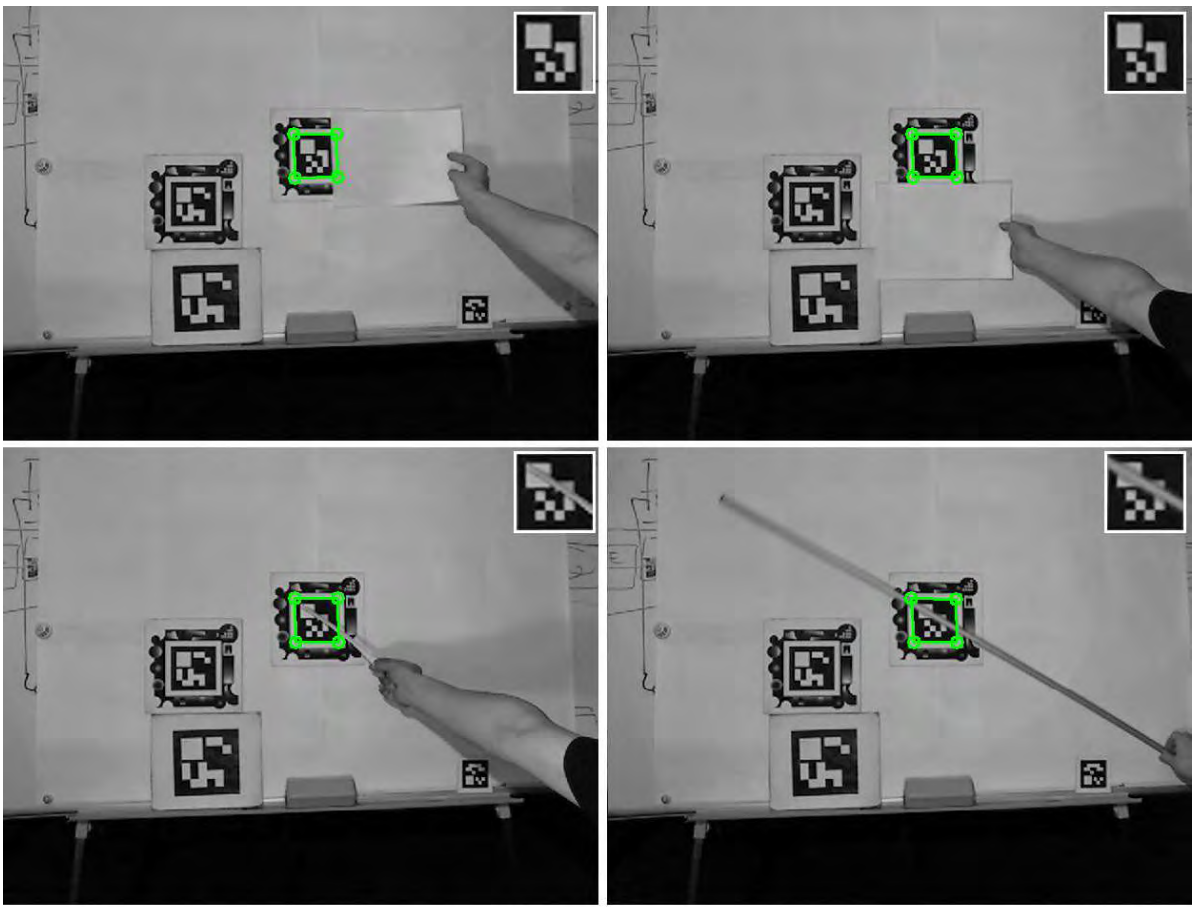


Figure 5: Examples of S-G method capability of occluded marker identification from a large distance. Both the marker boarder and marker content may be occluded.

In the angles comparison, measured results are influenced by the SURF algorithm limitations. The S-G method is able to detect a marker that is under 55° angle to the camera axis. (0° represents a marker perpendicular to the camera axis. The maximal theoretical angle is therefore 90°.) The other two solutions have maximal angles ranging from 74° to 85°.

Neither *ARToolKitPlus* nor *ALVAR* can deal with any type of occlusion. This is the most important disadvantage of these solutions. The S-G method can deal with significant marker occlusion. Because S-G works with key points instead of morphological operations or e.g. *edgels*, it is able to withstand a substantial number of different occlusions. We tested several of them (see Fig. 4).

The marker border can be obstructed up to 50 %. It is irrelevant what part of marker border is obstructed (all corners, two whole sides, etc.). The marker content (the Golay error correction code) must be visible at least from 75 % in case the large white square is obstructed. In case the obstruction is in other part of the Golay code, maximum allowed occlusion is approx. 15 %. This occlusion is limited by the Golay code redundancy.

This is the most important contribution of our solution in comparison to other used methods**.**

Because of the nature of the detection, the solutions capable of occlusion (e.g. *ARTag*) need at least three visible marker corners to detect and identify the marker. Our method is capable of the identification of a marker with all corners or sides covered. Our method has capability even of overcoming of the occlusion of marker contents. This is possible because of the Golay error correction code usage.

Figure 6: Marker occlusion. The marker is approx. 2 m distant from the camera.

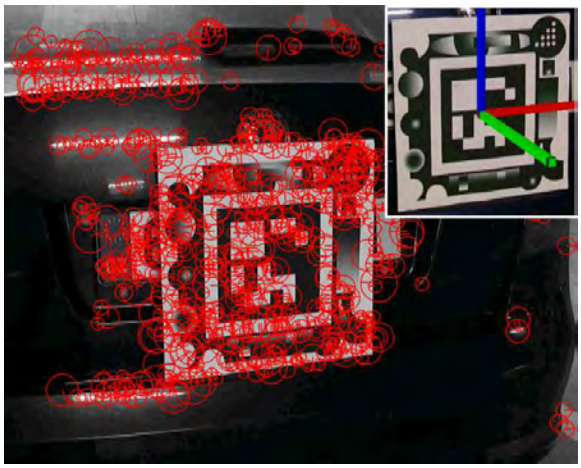


Figure 7: Key points detected by the S-G method and augmented 3D model.

5. CONCLUSION

Our application aims to improve the car design process. Therefore, several criteria must be fulfilled: Our marker detection and identification methods must be able to distinguish several hundred markers (one marker represents one spare part). Further, it must be possible to compute the precise position and rotation of the marker. Finally, the methods must be able to deal with occlusions that are common in real situations.

The SURF detection method as well as the Golay error correction code is able to deal with the occlusions. The proposed S-G registration method is slower than other frequently used approaches (e.g. image morphology approach with the Golay error correction codes). Still, it works in a constant time that is significant for real-time applications.

Nevertheless, in case of very good lighting conditions and absence of occlusions we recommend techniques based on the image morphology. With these methods, the video stream processing speed is substantially improved. Our *AuRel* application supports both approaches; therefore, registration technique is chosen according to the current conditions. By default, the morphology-based method (16 fps) is used. In case a marker detected in previous frame is missing, we switch to the S-G method (4 fps). Following frame is again processed by morphology-based method. Frame rates are measured on 640×480 px camera stream processed by Intel Core i5 2.6 GHz, 4 GB RAM, HDD 7200 rpm.

We consider our approach very promising. Nonetheless, there must be further research focused on several technical aspects. Particularly, the marker detector performance should be optimized (on the reference hardware configuration, *ARToolKitPlus* and *ALVAR* have above 20 fps). This could be done by reducing the number of key points in exchange for

lower reliability. Also the maximum detection distance needs to be improved. Possible solution can be to improve marker design so that the marker detector response is increased as outlined in [Sch*09].

SURF method can be easily used to design a marker-less tracking method as outlined in many articles. The absence of markers can substantially improve the application usability. Nevertheless, there could be a problem with selection of a correct 3D model and its manual position adjustment.

6. ACKNOWLEDGMENTS

This paper is written as a part of a solution of the project IGA FBE MENDELU 7/2012 and FBE MENDELU research plan MSM 6215648904.

7. REFERENCES

[BCP*08] Barandiaran, I., Cottez, Ch., Paloc, C., Grana, M.: Comparative Evaluation of Random 159 Forest and Ferns Classifiers for Real-Time Feature Matching in WSCG 2008 Full Papers Proceedings, Plzen: University of West Bohemia, 2008, pp. 159-166.

[Bru09] Brunelli, R.: Template Matching Techniques in Computer Vision: Theory and Practice, Wiley, 2009, ISBN 978-0-470-51706-2.

[BKF*00] Balcisoy, S., Kallmann, M., Fua, P., Thalmann, D.: A framework for rapid evaluation of prototypes with Augmented Reality. In Proceedings of the ACM symposium on Virtual reality software and technology, pp. 61-66. 2000. ISBN:1-58113-316-2.

[BTG06] Bay, H., Tuytelaars, T., Gool, L. V.: Surf: Speeded up robust features. In ECCV, 2006, pp. 404–417.

[CHT*09] Cui, Y., Hasler, N., Thormahlen, T., Seidel, H.: Scale Invariant Feature Transform with Irregular Orientation Histogram Binning. In Proceedings of Image Analysis and Recognition: 6th International Conference, pp. 258-267, 2009. ISBN: 978-3-642-02610-2.

[FAM*02] Fiorentino, M., De Amicis, R., Monno, G., Stork, A.: Spacedesign: A Mixed Reality Workspace for Aesthetic Industrial Design. In Proceedings of International Symposium on Mixed and Augmented Reality, p. 86. 2002. ISBN:0-7695-1781-1.

[Fia05] Fiala, M.: ARTag, a fiducial marker system using digital techniques. Computer Vision and Pattern Recognition, 2, June, 2005.

[Hir08] Hirzer, M.: Marker detection for augmented reality applications, 2008. Inst. for Computer Graphics and Vision, Graz University of Technology, Austria.

[HNL96] Hoff, W. A., Nguyen, K., Lyon T.: Computer vision-based registration techniques for augmented reality. In Intelligent Robots and Computer Vision, XV, 1996, pp. 538–548.

[KB99] Kato, H., Billinghurst, M.: Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System. In Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality. 1999, s. 85-94.

[KTB*03] Kato, H., Tachibana, K., Billinghurst, M., Grafe, M.: A registration method based on texture tracking using artoolkit. In Augmented Reality Toolkit Workshop, 2003. IEEE International, 2003, IEEE, pp. 77–85.

[Lag11] Laganiere, R.: OpenCV 2 Computer Vision Application Programming Cookbook. Packt Publishing, 2011.

[Low04] Lowe, D. G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision, 60, 2004, pp. 91–110.

[MZ06] Morelos-Zaragoza, R. H.: The Art of Error Correcting Coding (Second Edition). John Wiley & Sons, 2006.

[PK11] Prochazka, D., Koubek, T.: Augmented Reality Implementation Methods in Mainstream Applications. Acta of Mendel University of agriculture and forestry Brno 59, 4,. 2011, p. 257.

[Sch*09] Schweiger, F. et al.: Maximum Detector Response Markers for SIFT and SURF, Proceedings of the Vision, Modeling, and Visualization Workshop 2009, Germany.

[SZG*09] Schweiger, F., Zeisl, B., Georgel, P., Schroth, G., Steinbach, E., Navab, N.: Maximum Detector Response Markers for SIFT and SURF. In Int. Workshop on Vision, Modeling and Visualization (VMV), 2009.

[Sze11] Szeliski, R.: Computer Vision: Algorithms and Applications, Springer, 2011.

[TM08] Tuytelaars T., Mikolajczyk K.: Local invariant feature detectors: a survey, Foundations and Trends® in Computer Graphics and Vision archive, Volume 3 Issue 3, 2008, pp. 177-280.

[VSP*03] Verlinden, J. C., Smit, A. D., Peeters, A. W. J., Gelderen, M. H. V.: Development of a flexible augmented prototyping system. Journal of WSCG, 11, 2003, pp. 496-503

A Survey of Cloud Lighting and Rendering Techniques

Roland Hufnagel
Univ. Salzburg, Salzburg, Austria
rhufna@cosy.sbg.ac.at

Martin Held
Univ. Salzburg, Salzburg, Austria
held@cosy.sbg.ac.at

ABSTRACT

The rendering of participating media still forms a big challenge for computer graphics. This remark is particularly true for real-world clouds with their inhomogeneous density distributions, large range of spatial scales and different forms of appearance. We survey techniques for cloud visualization and classify them relative to the type of volume representation, lighting and rendering technique used. We also discuss global illumination techniques applicable to the generation of the optical effects observed in real-world cloud scenes.

Keywords

cloud rendering, cloud lighting, participating media, global illumination, real-world clouds

1 INTRODUCTION

We review recent developments in the rendering of participating media for cloud visualization. An excellent survey on participating media rendering was presented by Cerezo et al. [5] a few years ago. We build upon their survey and present only recently published techniques, with a focus on the rendering of real-world cloud scenes. In addition, we discuss global illumination techniques for modeling cloud-to-cloud shadows or inter-reflection.

We start with explaining real-world cloud phenomena, state the graphics challenges caused by them, and move on to optical models for participating media. In the following sections we categorize the state-of-the-art according to three aspects: The representation of clouds (Section 2), rendering techniques (Section 3) and lighting techniques (Section 4).

1.1 Cloud Phenomenology

Clouds exhibit a huge variety of types, differing according to the following aspects.

Size: Clouds reside in the troposphere, which is the layer above the Earth's surface reaching up to heights of 9–22 km. In the mid-latitudes clouds show a maximum vertical extension of 12–15 km. Their horizontal extension reaches from a few hundred meters (Fig. 1.7) to connected cloud systems spanning thousands of kilometers (Figs. 1.11 and 1.12).

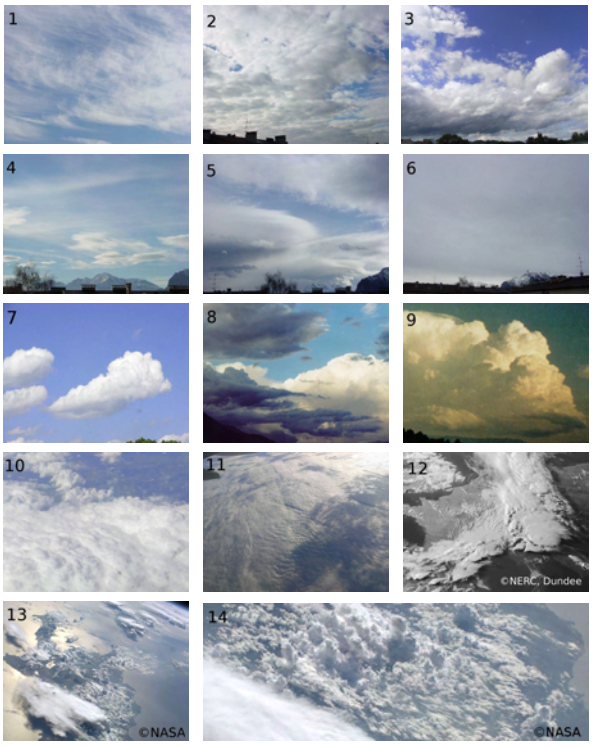


Figure 1: Clouds as seen from the ground: broken cloud layers with fractal cloud patches (top row), clouds with diffuse boundaries (second row), dense cloud volumes with surface-like boundaries (third row); and clouds viewed from a plane and from space (last two rows), where (14) shows a zoom into the central region of (13).

Geometry: In relation to the Earth's radius (6371 km) the troposphere represents a shallow spherical shell. The curvature of the Earth produces the horizon and is directly visible when viewing clouds from space.

Clouds often develop at certain heights and form layers. These either consist of cloud patches (Fig. 1.1 to 1.5) or create overcast cloud sheets (Fig. 1.6 or 1.11).

Local upward motions emerging from the ground (convective plumes) create clouds with a sharp cloud base and a cauliflower-like structure above, so-called Cumulus (Figs. 1.7 to 1.9). Their vertical extension reaches up to several kilometers (Fig. 1.9).

For low clouds also mixed forms of convective and layered clouds exist, where the convective plumes are vertically bound and form a layer (Fig. 1.3).

Clouds formed in connection with frontal systems usually show a large vertical extension and can span thousands of kilometers (Fig. 1.12).

Boundary Appearance: The appearance of clouds mainly depends on the cloud volume's constituents: droplets of different size or ice particles. While large water droplets produce a determined, surface-like boundary (Figs. 1.7 to 1.9), smaller droplets create a diffuse or fractal boundary (Figs. 1.2 to 1.6). Ice particles often form hair-like or fiber-like clouds, so-called Cirrus (Fig. 1.1), or diffuse clouds, the anvil-like tops of convective clouds (Fig. 1.13).

The appearance of a cloud is strongly influenced by the distance to its observer: While distant clouds often show a distinct surface and sharp contours (Figs. 1.12 and 1.13), a closer look reveals diffuse or fractal structures (Fig. 1.14).

Optical Phenomena: Clouds consist of water droplets and ice crystals which scatter light mostly independent of wavelength. Clouds are therefore basically white. Spectral colors appear only at certain angular constellations but generally do not influence their overall appearance. However, several optical phenomena determine their characteristic, natural appearance:

Self-Shadowing: The attenuation of light within a cloud creates gray tones and is proportional to the optical depth of the volume. The self-shadowing provides the cue to perceive clouds as volumetric objects.

Multiple scattering slightly attenuates the effect of self-shadowing by distributing light within the cloud volume in a diffusion-like process; see e.g. [4, 27, 33].

Inter-Cloud Shadows: Clouds cast shadows onto other clouds, like in Fig. 1.11, where a high cloud layer on the right-hand side shadows a low cloud layer.

The Earth's Shadow: Clouds can be shadowed by the Earth; see Fig. 1.8 showing an evening scene, where the low clouds lie in the shadow of a mountain range.

Indirect Illumination: Light inter-reflection between different clouds or between different parts of the same cloud brighten those regions, as, e.g., in Fig. 1.9, where the cloud seems to gloom from the inside.

Light Traps: Light inter-reflection at a smaller scale occurs on determined cloud surfaces (Neyret [29]) and lets concavities appear brighter (Figs. 1.7 and 1.10).

Corona: The corona effect occurs when the cloud is lit from behind. The strong forward scattering at the boundary produces a bright silhouette (silver-lining).

Atmospheric Scattering: Clouds often appear in vivid colors. This is caused by the scattering of light outside the cloud volume on air molecules and aerosols. Clouds are therefore often lit by yellowish to reddish sunlight (Fig. 1.9). Different paths of light in the atmosphere let the high clouds in Fig. 1.14 appear bright white and the low clouds yellowish. Atmospheric scattering also creates blue skylight which in some situations represents the main source of lighting (Fig. 1.8).

Ground Inter-Reflection: For low-level clouds the inter-reflection with the ground creates subtle tones depending on the type of the ground; see e.g. [3].

1.1.1 Summary

From a computer graphics point of view we identify the following volume properties, in addition to the form of the clouds as created by cloud modeling techniques, which are out of the scope of this survey: **thin** clouds that show no self-shadowing; cloud patches that represent a mixture of slightly dense cores and optically thin boundaries, usually forming horizontally extensive **layered** clouds; and **dense** cloud volumes of different sizes and extensions with a sharp or surface-like boundary. The boundary of clouds exhibits either a **fractal**, **diffuse** or **sharp** appearance.

1.2 Computer Graphics Challenges

A realistic visualization of clouds requires to tackle the following challenges:

Heterogeneity: Real-world cloud scenes typically consist of a very heterogeneous collection of clouds with different appearances, sizes and forms. Different cloud types require different volume representations, lighting and rendering techniques.

Atmospheric Scattering: For creating a cloud's natural appearance the lighting model employed has to reproduce its typical optical phenomena (see Sec. 1.1), including atmospheric scattering which is the main source for color in the sky. This requires the inclusion of atmospheric models (which are not discussed in this survey) in the lighting process of clouds.

Curved volume: The spherical atmosphere makes it difficult to take advantage of axis-aligned volumes if clouds are viewed on a large or even global scale, as in Figs. 1.12 or 1.13.

Huge domain: The sheer size of the volume of real-world cloud scene, especially when viewed from above, like in Figs 1.11 to 1.13, requires sophisticated and efficient lighting and rendering techniques.

1.3 Participating Media

A cloud volume constitutes a participating medium exhibiting light attenuation and scattering. This section uses the terminology of [5] to provide a short introduction to the radiometry of participating media.

While light in vacuum travels along straight lines, this is not the case for participating media. Here photons interact with the medium by being scattered or absorbed. From a macroscopic point of view light spreads in participating media, gets blurred and attenuated, similar to heat diffusion in matter.

Participating media are characterized by a particle density ρ, an absorption coefficient κ_a, a scattering coefficient κ_s, and a phase function $p(\vec{\omega},\vec{\omega}')$ which describes the distribution of light after scattering.

Absorption is the process where radiation is transformed to heat. The attenuation of a ray of light with radiance L and direction $\vec{\omega}$ at position x (within an infinitesimal ray segment) is described by

$$(\vec{\omega}\cdot\nabla)L(x,\vec{\omega}) = -\kappa_a(x)\,L(x,\vec{\omega}).$$

In the atmosphere absorption is mainly due to water vapor and aerosols. Cloud droplets or ice crystals show little absorption which means that the light distribution in clouds is dominated by scattering.

Scattering is the process where radiance is absorbed and re-emitted into other directions. *Out-scattering* refers to the attenuation of radiance along direction $\vec{\omega}$ due to scattering into other directions:

$$(\vec{\omega}\cdot\nabla)L(x,\vec{\omega}) = -\kappa_s(x)\,L(x,\vec{\omega}).$$

In-scattering refers to the scattering of light into the direction $\vec{\omega}$ from all directions (integrated over the sphere) at a point x:

$$(\vec{\omega}\cdot\nabla)L(x,\vec{\omega}) = \frac{\kappa_s(x)}{4\pi}\int_{4\pi} p(\vec{\omega}',\vec{\omega})\,L(x,\vec{\omega}')\,d\omega',$$

Extinction is the net effect of light attenuation due to absorption and out-scattering described by the extinction coefficient $\kappa_t = \kappa_a + \kappa_s$.

Emission contributes light to a ray:

$$(\vec{\omega}\cdot\nabla)L(x,\vec{\omega}) = \kappa_a(x)\,L_e(x,\vec{\omega}).$$

It is usually not relevant for cloud rendering since clouds do not emit light. (An exception is lightning inside a cloud.)

Radiative Transfer Equation (RTE): Putting all terms together yields the RTE which describes the change of radiance within a participating medium at a point x:

$$\begin{aligned}(\vec{\omega}\cdot\nabla)L(x,\vec{\omega}) \;=\;& \kappa_a(x)\,L_e(x,\vec{\omega}) + \\ & \frac{\kappa_s(x)}{4\pi}\int_{4\pi} p(\vec{\omega}',\vec{\omega})\,L(x,\vec{\omega}')\,d\omega' - \\ & \kappa_a(x)\,L(x,\vec{\omega}) - \kappa_s(x)\,L(x,\vec{\omega}).\end{aligned}$$

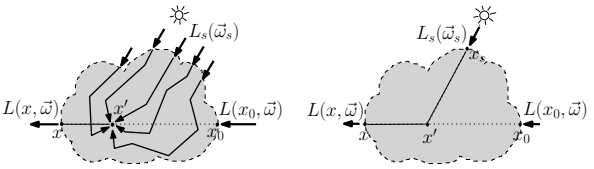


Figure 2: Multiple scattering (left), and the single scattering approximation (right).

By using the single scattering albedo $\Omega = \kappa_s/\kappa_t$ and noting that κ_a can be expressed as $\kappa_a = \kappa_t(1-\Omega)$, we can re-write the RTE as

$$(\vec{\omega}\cdot\nabla)L(x,\vec{\omega}) = \kappa_t(x)\,J(x,\vec{\omega}) - \kappa_t(x)\,L(x,\vec{\omega}), \quad (1)$$

with the source radiance J:

$$\begin{aligned}J(x,\vec{\omega}) \;=\;& (1-\Omega(x))\,L_e + \\ & \frac{\Omega(x)}{4\pi}\int_{4\pi} p(\vec{\omega}',\vec{\omega})\,L(x,\vec{\omega}')\,d\omega'.\end{aligned}$$

The source radiance describes all contributions of radiance to a ray $(x,\vec{\omega})$ at a point x inside the medium. In high-albedo media, like clouds, the source term is mainly due to in-scattering, while the extinction is dominated by out-scattering: $\kappa_t \approx \kappa_s$.

Integrating the RTE along a ray from x_0 to x yields the radiance reaching point x from direction $-\vec{\omega}$ (see Fig. 2, left):

$$L(x,\vec{\omega}) = T(x,x_0)L(x_0,\vec{\omega}) + \int_{x_0}^{x} T(x,x')\,\kappa_t(x')\,J(x')\,dx', \quad (2)$$

with the transmittance $T(x_1,x_2) = \exp(-\int_{x_1}^{x_2}\kappa_t(x)\,dx)$. The boundary condition of the integral is $L(x_0,\vec{\omega})$, representing the light coming from the background, an environment map, or from scene objects.

Note that the coefficients κ_* depend on the wavelength. Therefore, three versions of Eqn. 2 have to be solved with appropriate coefficients $\kappa_{*,\lambda}$ for each wavelength corresponding to the RGB color components.

Single Scattering Approximation: A difficulty in solving Eqn. 2 is that L appears (implicitly through J) on both sides of the equation. A common approximative solution is to account only for a certain number of scattering events and apply extinction on the paths in between. Considering only the first order of scattering yields the single scattering approximation: The source radiance J_{SS} is given by the light from the light source L_s attenuated on its way between x_s and x', see Fig. 2 (right), thus eliminating L:

$$J_{SS}(x',\vec{\omega}) = \Omega(x')\,T(x',x_s)\,p(x',\vec{\omega}_s,\vec{\omega})\,L_s(x_s,\vec{\omega}_s).$$

The single scattering approximation simulates the self-shadowing of a volume. Higher order scattering accounts for the "more diffuse" distribution of light within the volume.

Phase Function: A scattering phase function is a probabilistic description of the directional distribution of scattered light. Generally it depends on the wavelength, and the form and size of the particles in a medium. Usually phase functions show a symmetry according to the incident light direction, which reduces it to a function of the angle θ between incident and exitant light $p(\theta)$.

Often approximations for certain types of scattering are used, like the *Henyey-Greenstein* or the *Schlick* function. However, as noted in [4], those functions cannot model visual effects that depend on small angular variations, like glories or fog-bows. See [4] and its references for plots and tabular listings of phase functions.

2 CLOUD REPRESENTATIONS

A cloud representation specifies the spatial distribution, overall structure, form, and boundary appearance of clouds in a cloud scene.

2.1 Hierarchical Space Subdivision

2.1.1 Voxel Octrees

Voxel octrees are a hierarchical data structure built upon a regular grid by collapsing the grid cells of a $2 \times 2 \times 2$ cube (children) to a single voxel (parent).

Sparse voxel octrees reduce the tree size by accounting for visibility and LOD: Interior voxels are removed, yielding a hull- or shell-like volume representation (see Fig. 3, middle). Also hidden voxels (relative to the current viewing point) are removed (see Fig. 3, right). Additionally the LOD resolution can be limited according to the screen resolution (view-dependent LOD). These techniques require an adaptive octree representation accompanied with an update strategy.

Crassin et al. [7], and similarly Gobetti et al. [13], propose a dynamic octree data structure in combination with a local regular grid representation. Each node of the octree is associated with a *brick*, a regular 32^3 voxel grid, which represents a filtered version of the volume enclosed by its child nodes. The bricks are stored in a *brick pool* of a fixed size. Bricks are referenced by pointers stored in the nodes of the octree (see Fig. 4). The octree nodes themselves are stored in a pool as well (*node pool*). During the visualization brick and node data is loaded on demand and the pools are managed by updating least recently used data.

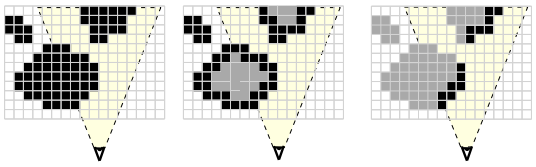


Figure 3: Voxel volume (left), shell-like boundary voxels (middle), culling invisible voxels (right).

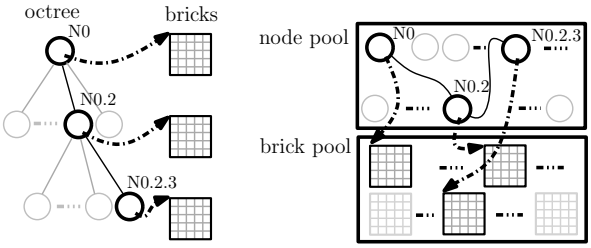


Figure 4: Sketch of the GigaVoxel data structure [7].

Laine and Karras [25] store an octree data structure in a single pool. The data is divided into blocks which represent contiguous areas of memory allowing local addressing and taking advantage of fast memory access operations on the GPU (caching). The data structure is designed to compactly store mesh-based scenes but their open-source implementation could probably be adapted to represent cloud volumes.

Miller et al. [28] use a grid representation on the coarse scale and nest octrees within those cells. This provides fast grid marching on a large scale, and adaptive sampling on the small scale. A fixed data structure, resembling 4-level octrees, allows to directly access the octree's nodes without requiring pointers. However, their current implementation assumes that the whole scene fits into the memory of the graphics device, which limits the model size. A streaming-based data structure for dynamic volumes was announced as future work.

The real-time voxelization of scene geometry, as proposed by Forester et al. [12], allows to transform rasterizable geometry to an octree representation on-the-fly on the GPU, and thus to apply voxel-based lighting and rendering to a surface-based geometry.

Octrees are a flexible data structure, generally capable of representing all types of clouds and supporting efficient lighting and rendering techniques. However, since octrees usually take advantage of axis-aligned grids, they are not directly applicable in an efficient way to large-scale cloud scenes, but would have to be nested in the spherical shell or used with a ray caster that takes into account the curvature of the Earth.

2.1.2 Binary Space Partitioning (BSP)

BSP, e.g., in form of kd-trees, recursively subdivides the space into half-spaces, concentrating at regions with high geometric detail and removing empty space. BSP could be used for cloud volume representation as well.

2.1.3 Bounding Volume Hierarchies (BVH)

BVH enclose scene geometry by bounding planes. Recently, BVH were used for structuring particle systems [14], which could also be employed for cloud volumes.

2.2 Implicit Representations

A common way to model and represent a cloud's density field is the use of procedural methods. While the

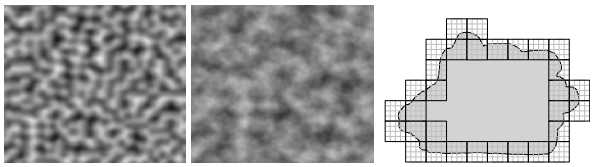


Figure 5: Perlin noise [31] and "fractal sum" [11] functions (left). A hypertexture applied to a surface (right).

overall structure is usually specified by simple geometric primitives, like spheres or ellipsoids, the internal, high-resolution structure is modeled by a function.

Perlin and Hoffert [31] introduce space-filling shapes, based on procedural functions, so-called *hypertextures*. Ebert et al. [11] propose many additional noise functions (see Fig. 5, left), and create various natural patterns applicable also for clouds.

The sampling of implicitly defined densities can become expensive and harm rendering performance. Schpok et al. [36] propose to evaluate the procedural functions on-the-fly on the GPU during the rendering by using a fragment shader program and a 3D texture containing Perlin noise.

Kniss et al. [22] use procedural functions for geometric distortion which adds a fractal appearance to regular shapes by changing the vertex positions of the geometry rendered. They apply this distortion during the rendering process by using vertex shader programs.

Bouthors et al. [4] use hypertextures in combination with surface-bounding volumes for creating a fractal boundary appearance; see Fig. 5, right.

Implicitly specifying a volume via procedural functions is a compact volume representation. The computational cost can be mitigated by employing parallel processing on the GPU and taking advantage of hardware-supported tri-linear interpolation of 3D textures. Procedural techniques are perfect for clouds with a fractal boundary appearance. However, they only provide the fine-scale volume structure while the overall cloud shape has to be modeled by other techniques.

2.3 Particle Systems

The volume is represented by a set of particles with a pre-defined volume. Usually spherical particles with a radial density function are used.

Nishita et al. [30] promote the use of particles with a Gaussian density distribution, so-called *metaballs*. While the metaballs in [30] are used only to create a volume density distribution, Dobashi et al. [10] directly light and render the metaballs and visualize medium-size cloud scenes of Cumulus-like clouds with a diffuse boundary appearance.

Bouthors and Neyret [2] use particles to create a shell-like volume representation for Cumulus-like clouds. Their algorithm iteratively places particles at the interface of a cloud volume, with smaller particles being placed upon the interface of larger particles. This creates a hierarchy of particles with decreasing radius and specifies the cloud's surface, which can be transformed, e.g., to a triangle mesh.

Efficient transformation algorithms were developed to match the favored lighting and rendering approaches. Cha et al. [6] transform the density distribution given by a particle system to a regular grid by using the GPU, while Zhou et al. [44] propose the inverse process transforming a density field to *radial basis function* (RBF) representation. This low-resolution particle system is accompanied by a high-resolution grid, which stores deviations of the particles' density distribution from the initial density field in a so-called *residual field*. *Perfect spatial hashing* allows a compact storage of this field.

Particles systems are a compact volume representation and directly support many cloud modeling techniques. Spherical particles are well suited for Cumulus-like clouds or dense cloud volumes, but less appropriate for stratified cloud layers with a large horizontal extension or for thin, fiber-like clouds.

2.4 Surface-Bounded Volumes

The cloud volume is represented by its enclosing hull, usually given as a triangle mesh. Since no information on the internal structure is available, usually a homogeneous volume is assumed.

Bouthors et al. [4] demonstrate the use of surface-bounded volumes for visualizing single Cumulus clouds. A triangle mesh is used in combination with a hypertexture to add small-scale details at the boundaries. A sophisticated lighting model reproduces a realistic appearance (see Sec. 4.1.4).

Porumbescu et al. [32] propose *shell maps* to create a volumetric texture space on a surface. A tetrahedral mesh maps arbitrary volumetric textures to this shell.

Surface-bounded volumes are a very compact representation of cloud volumes, allowing for efficient rendering and for incorporating sophisticated lighting techniques. However, they are only applicable if a quick saturation of a ray entering the volume may be assumed. While this is valid for dense clouds it does not apply to thin or layered clouds with their optically thin boundaries. Also special rendering techniques have to be developed for allowing perspectives from within the clouds.

2.5 Clouds as Layers

Clouds represented by a single layer, usually rendered as a textured triangle mesh, allow fast rasterization-based rendering and are the traditional technique to present clouds, e.g., during weather presentations [24].

Bouthors et al. [3] visualize cloud layers viewed from the ground or from above. They achieve a realistic appearance of the cloud by applying a sophisticated, viewpoint-dependent lighting model.

The 2D representation is especially predestined for thin cloud sheets viewed from the ground, or for visualizing the cloud tops viewed, e.g., from the perspective of a satellite. However, this non-volumetric representation limits the possible perspectives and generally does not allow for animations, like transitions of the viewpoint from space to the ground, or cloud fly-throughs.

3 CLOUD RENDERING TECHNIQUES

3.1 Rasterization-based Rendering

3.1.1 Volume Slicing

Volume slicing is a straightforward method for rendering regular grids. The slices are usually axis aligned and rendered in front-to-back order (or vice-versa), applying viewing transformations and blending. While volume slicing is not necessarily a rasterization-based method, most algorithms exploit the highly optimized texturing capabilities of the graphics hardware.

Schpok et al. [36] use volume slicing for cloud rendering and propose the use of the GPU also for adding detailed cloud geometry on-the-fly by using a fragment shader program. This reduces the amount of data which has to be transferred onto the GPU to a low-resolution version of the cloud volume. Harris et al. [15] create the density volume by a CFD simulation on the GPU for creating a 3D density and light texture which can efficiently be rendered on the GPU. Sparing the transfer of the volumetric data from the CPU, they achieve interactive frame rates for small volumes (up to 64^3), creating soft, diffuse clouds. Hegeman et al. [17] also use 3D textures to capture the volume density and the pre-calculated source radiance, and evaluate a lighting model in a CG shader program during rendering.

Zhou et al. [44] visualize smoke represented by a low-resolution particle system accompanied by a high-resolution density grid, stored in compressed form (see Sec. 2.3). During the rendering process the lighted particles are first rasterized and converted to a 3D texture by applying parallel projection rendering along the z-axis to fill slices of the 3D texture. Thereby, for each slice, all particles are traversed and, in case of intersection with the slicing plane, rendered as textured quads. During this pass also the residual field is evaluated and stored in a separate 3D texture. In a second step perspective rendering is applied by slicing in back-to-front manner. Again, for each slice all particles are traversed and bounding quads are rendered triggering a fragment shader which composes the light and density information from the 3D textures. They achieve interactive frame rates under dynamic lighting conditions for moderate volume sizes which completely fit into the GPU's memory (128^3 in their examples), based on 0.5 to 1.5 hours of pre-processing time.

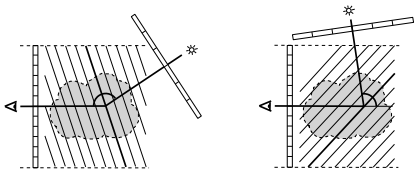


Figure 6: Volume rendering by half-angle slicing.

Half-Angle Slicing: Slicing the volume at planes oriented halfway between the lighting and viewing direction (or its inverse, see Fig. 6) is called half-angle slicing. It allows to combine the lighting and rendering of the volume in a single process by iterating once over all slices. During this single-volume pass two buffers are maintained and iteratively updated: one for accumulating the attenuation of radiance in the light direction, and one for accumulating the radiance for the observer (usually in the frame buffer). Due to the single pass through the volume, the lighting scheme is limited to either forward or backward scattering.

Kniss et al. [22] use half-angle slicing in combination with geometric distortion, modifying the geometry of shapes on-the-fly during rendering (see Sec. 2.2) by using vertex shader programs. Riley et al. [35] visualize thunderstorm clouds, taking into account different scattering properties of the volume.

For avoiding slice-like artifacts volume slicing techniques have to employ small slicing intervals, resulting in a large number of slices which can harm rendering performance and introduce numerical problems. Inhomogeneous volumes, like in Figs. 1.2 or 1.14, would also require a huge number of slices for appropriately capturing the volume's geometry and avoiding artifacts in animations with a moving viewpoint. Slicing-based methods therefore seem to favor volumes with soft or diffuse boundaries and, thus, are applicable to clouds with sharp boundaries only to a limited extent.

3.1.2 Splatting

Splatting became the common method for rendering particle systems. Particles, which are usually specified as independent of rotation, can be rendered by using a textured quad representing the projection of the particle onto a plane, also called splat or footprint. The particles are rendered in back-to-front order, applying blending for semi-transparent volumes.

Dobashi et al. [10] use metaballs for splatting cloud volumes. Harris et al. [16] accelerate this approach by re-using impostors, representing projections of a set of particles, for several frames during fly-throughs. The use of different particle textures can enhance the clouds' appearance [42, 18].

Since usually a single color (or luminance) is assigned to each particle and the particles do not represent a distinct geometry, but a spherical, diffuse volume (due to the lack of self-shadowing within a particle), the splatting approach is limited to visualizing clouds with a soft, diffuse appearance. Clouds with a distinct surface geometry, like Cumulus clouds, cannot be reproduced realistically.

3.1.3 Surface-Based Volume Rendering

Nowadays, rendering clouds as textured ellipsoids is no more regarded as realistic. The same applies to the surface-bounded clouds of [41]. A triangle mesh is created by surface subdivision of an initial mesh, created by a marching cubes algorithm applied on weather forecast data. The rendering of the semi-transparent mesh, however, is prone to artifacts on the silhouette.

Bouthors et al. [4] resurrected surface-based cloud rendering by using fragment shader programs which calculate the color for each fragment of a triangle mesh at pixel-basis. This allows to apply a sophisticated volume lighting scheme and the sampling of a hypertexture superposed onto the surface on-the-fly for each pixel. They achieve a realistic, real-time visualization of dense clouds with fractal and sharp boundaries.

3.2 Ray Casting-Based Rendering

3.2.1 Ray Marching

Ray marching casts rays into the scene and accumulates the volume densities at certain intervals. For rendering participating media volumes, like clouds, the illumination values of the volume have to be evaluated, either by applying a volume lighting model on-the-fly or by retrieving the illumination from a pre-computed lighting data structure.

Grids: Cha et al. [6] transform a particle-based volume representation to a regular density grid for applying ray marching. The volume density is sampled within a 3D texture, combined with the illumination, stored in a separate 3D texture, and accumulated along the viewing rays. Geometric detail is added to the low-resolution volume representation on-the-fly by slightly distorting the sampling points of the ray march according to a precomputed 3D procedural noise texture. For volumes fitting into the memory of the GPU (around 256^3 in their examples), they achieve rendering times of a few seconds (including the lighting calculation).

BVHs: A particle system stored within a kd-tree structure is proposed by Gourmel et al. [14] for fast ray tracing. The technique could be used for cloud rendering, e.g., by ray marching through the particle volumes and adding procedural noise as proposed in [4] or [6].

Octrees: The huge amount of data caused by volumetric representations can be substantially reduced by using ray-guided streaming [7, 13, 25]. The GigaVoxel algorithm [7] is based on an octree with associated bricks (Sec. 2.1.1) which are maintained in a pool and streamed on demand. The bricks at different levels of the octree represent a multi-resolution volume, similar to a mip-map texture. Sampling densities at different mip-map resolutions simulates cone tracing (see Fig. 4, right). The octree is searched in a stack-less manner, always starting the search for a sampling position at the root node. This supports the cone-based sampling of the volume since the brick values at all levels can be collected during the descent. The implementation employs shader programs on the GPU, and achieves 20–90 fps, with volumetric resolutions up to 160k.

4 LIGHTING TECHNIQUES

4.1 Participating Media Lighting

We review recent volume lighting approaches and refer to [5] for a survey of traditional, mainly off-line lighting models.

4.1.1 Single-Scattering Approximation

Slice-Based: Schpok et al. [36] use volume slicing for creating a low-resolution volume storing the source radiances. This so-called light volume is oriented such that light travels along one of the axes, which allows a straightforward light propagation from slice to slice. The light volume storing the source radiance is calculated on the CPU and transferred to a 3D texture on the GPU for rendering.

In the half-angle slicing approach by Kniss et al. [22, 23] the light is propagated from slice to slice through the volume by employing the GPU's texturing and blending functionalities. A coarser resolution can be used for the 2D light buffer (Fig. 6), thus saving resources and accelerating the lighting process.

Particle-Based: Dobashi et al. [10] use shadow casting as a single-scattering method for lighting a particle system. The scene is rendered from the perspective of the light source, using the frame buffer of the GPU as a shadow map. The particles are sorted and processed in front-to-back manner relative to the light source. For each particle the shadow value is read back from the frame buffer before proceeding to the next particle and splatting its shadow footprint. This read-back operation forms the bottleneck of the approach which limits either the model size or the degree of volumetric detail. Harris and Lastra [16] extend this approach by simulating multiple forward scattering and propose several lighting passes for accumulating light contributions from different directions including skylight.

Bernabei et al. [1] evaluate for each particle the optical depth towards the boundary of the volume for a set

of directions and store it in a spherical harmonic representation. Each particle therefore provides an approximate optical depth of the volume for all directions, including the viewing and lighting directions (Fig. 8, left). The rendering process reduces to accumulating the light contributions of all particles intersecting the viewing ray, thus sparing a sorting of the particles, and provides interactive frame rates for static volumes. For the source radiance evaluation a ray marching on an intermediate voxel-based volume is used in an expensive pre-process. Zhou et al. [44] accomplish this in real-time by employing spherical harmonic exponentiation; see Sec. 4.1.6.

4.1.2 Diffusion

Light distribution as a diffusion process is a valid approximation in optically thick media, but not in inhomogeneous media or on its boundary. Therefore Max et al. [27] combine the diffusion with anisotropic scattering and account for cloudless space to reproduce, e.g., silver-lining. However, the computational cost is still high and causes rendering times of several hours.

4.1.3 Path Tracing

Path tracing (PT) applies a Monte Carlo approach to solve the rendering equation. Hundreds or even thousands of rays are shot into the scene for each pixel and traced until they reach a light source. PT produces unbiased, physically correct results but generally suffers from noise and low convergence rates. Only recent acceleration techniques allow an efficient rendering of large scenes, at least for static volumes.

In [39, 43], the volume sampling process of PT is accelerated by estimating the free path length of a medium in advance and by using this information for a sparse sampling of the volume ("*woodcock tracking*"). While Yue et al. [43] use a kd-tree for partitioning the volume, Szirmay-Kalos et al. [39] use a regular grid.

4.1.4 Path Integration

Path integration (PI) is based on the idea of following the path with the highest energy contribution and estimates the spatial and angular spreading of light along this so-called *most probable path* (MPP). PI favors high-order scattering with small scattering angles, and tends to underestimate short paths, diffusive paths and backward scattering.

The initial calculation model of Premoze et al. [33] is based on deriving a *point spread function* which describes the blurring of incident radiance. Hegeman et al. [17] proposed the evaluation of this lighting model using a shader program executed on the GPU. They visualize dynamic smoke (at resolutions up to 128^3) within a surface-based scene at interactive frame rates.

Bouthors et al. [4] use PI in combination with pre-computed tables which are independent of the cloud volume. In an exhaustive pre-computation process the impulse response along the MPP at certain points in a slab is calculated and stored as spherical harmonics (SH) coefficients (Fig. 7, left). During the rendering process the slabs are adjusted to the cloud volume (Fig. 7, right), and used for looking up the pre-computed light attenuation values. Multiplying it with the incident radiance at the cloud surface around the so-called collector area yields the resulting illumination. The model is implemented as a shader program which evaluates the lighting model pixel-wise in real-time.

4.1.5 Photon Mapping

Photon mapping (PM) traces photons based on Monte Carlo methods through a scene or volume and stores them in a spatial data structure that allows a fast nearest-neighborhood evaluation (usually a kd-tree). However, when applied to volumes the computational and memory costs are significant.

Cha et al. [6] combine photon tracing with irradiance caching, accumulating the in-scattered radiances at voxels of a regular grid. The gathering process reduces to a simple ray marching in the light volume executed on the GPU. This provides rendering times of a few seconds for medium-size volumes.

Progressive photon mapping as proposed by Knaus and Zwicker [21] improves traditional static photon mapping (PM) by reducing the memory cost and can also be applied to participating media. Photons are randomly traced through the scene, but instead of storing them in a photon map, they are discarded and the radiances left by the photons are accumulated in image space. Jarosz et al. [20] combine progressive PM with an improved sampling method of photon beams.

4.1.6 Mixed Lighting Models

Lighting techniques can be combined by evaluating different components of the resulting light separately.

Particle-Based: Zhou et al. [44] evaluate the source radiances at the particles' centers by accumulating the light attenuation of all particles onto each particle. Thereto a convolution of the background illumination function with the light attenuation function of the particles (spherical occluders) is applied (Fig. 8,

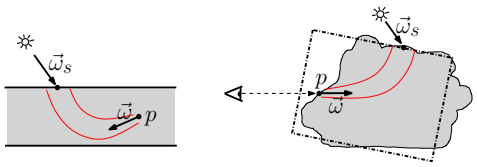


Figure 7: Pre-computed light in a slab (left), fitting slabs to a cloud surface (right).

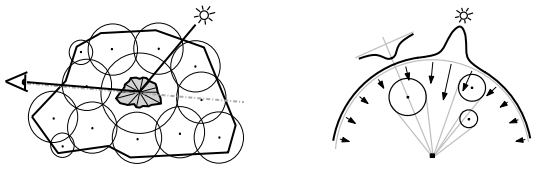


Figure 8: Pre-computed optical density at a particle center (left). Convolution of background illumination with the light attenuation by spherical occluders (right).

right). The functions are represented by low-order spherical harmonics (SH) and the convolution is done in logarithmic space where the accumulation reduces to summing SH coefficients; a technique called SH exponentiation (SHEXP), introduced by Ren et al. [34] for a shadow calculation. For multiple scattering an iterative diffusion equation solver is used on the basis of the source radiance distribution. This includes solving linear systems of dimension n, with n being the number of particles. The approximation of the smoke density field by a small number of particles (around 600 in their examples) allows interactive frame rates under dynamic lighting conditions. However, the super-quadratic complexity in terms of the particle number prohibits the application to large volumes.

Surface-Based: Bouthors et al. [3] separately calculate the single and multiple-scattering components of light at the surface of a cloud layer. The single-scattering model reproduces silver lining at the silhouettes of clouds and back-scattering effects (glories and fog-bows). Multiple-scattering is evaluated by using a BRDF function, pre-calculated by means of path tracing. Inter-reflection of the cloud layer with the ground and sky is taken into account by a radiosity model. They achieve real-time rendering rates by executing the lighting model on the GPU as a shader program.

4.1.7 Heuristic Lighting Models

Neyret [29] avoids costly physical simulations when a-priori knowledge can be used to simulate well-known lighting effects. He presents a surface-based shading model for Cumulus clouds that simulates inter-reflection in concave regions (light traps) and the corona at the silhouette.

Wang [42] does not employ a lighting model at all, but lets an artist associate the cloud's color to its height.

4.1.8 Acceleration Techniques

Pre-Computed Radiance Transfer: The lighting distribution for static volumes under changing lighting conditions can be accelerated by pre-computing the radiance transfer through the model.

Lensch et al. [26] pre-compute the impulse response to incoming light for a mesh-based model. Local lighting effects are simulated by a filter function applied to a light texture. Looking up the pre-computed vertex-to-vertex throughput yields global lighting effects.

Sloan et al. [38] use SH as emitters, simulate their distribution inside the volume and store them as transfer vectors for each voxel. The volume can be rendered efficiently in combination with a shader-based local lighting model under changing light conditions.

Radiance Caching: Jarosz et al. [19] use radiance caching to speed up ray marching rendering process. Radiance values and radiance gradients are stored as SH coefficients within the volume and re-used.

4.2 Global Illumination

Inter-cloud shadowing and indirect illumination cannot be efficiently simulated by participating media lighting models. These effects require global illumination techniques, usually applied in surface-based scenes.

4.2.1 Shadows

Sphere-based scene representations allow a fast shadow calculation, either, e.g., by using SH for representing the distribution of blocking geometries [34] (Fig. 8, right), or by accumulating shadows in image space [37].

The CUDA implementation of the GigaVoxel algorithm [8] allows to render semi-transparent voxels and to cast shadow rays. Employing the inherent cone tracing capability produces soft shadows.

4.2.2 Indirect Illumination

Voxel cone tracing applied to a sparse octree on the GPU is used by Crassin et al. [9] for estimating indirect illumination. The illumination at a surface point is evaluated by sampling the neighborhood along a small number of directions along cones.

A fast ray-voxel intersection test for an octree-based scene representation is used in Thiedemann et al. [40] for estimating near-field global illumination.

5 SUMMARY AND OUTLOOK

In the following table we summarize efficient lighting and rendering techniques for clouds with different types of volume (rows) and boundary appearances (columns).

	diffuse	fractal	sharp
dense	[6, 7, 10, 15] [16, 35, 44]	[4, 7, 11] [17, 22]	[4, 42]
thin	[22, 36, 44]	[11, 17, 36]	do not exist
layer	[3]		

The approaches surveyed represent a set of impressive but specialized solutions that employ fairly diverse techniques. However, all known approaches cover only some of the phenomena found in nature (see Sec. 1.1). Extensive memory usage or the algorithm's complexity often limit the size of a cloud volume. Thus, the realistic visualization of real-world cloud scenes still is and will remain widely open for future research for years to come.

Major future challenges to be tackled include the efficient handling of

- heterogeneous cloud scene rendering, i.e., the simultaneous visualization of different cloud types, each favoring a specific cloud representation, lighting and rendering technique;
- large-scale cloud scenes, e.g., clouds over Europe;
- cloud-to-cloud shadows and cloud inter-reflection, i.e., the combination of global illumination techniques with participating media rendering;
- the inclusion of atmospheric scattering models for lighting clouds;
- cloud rendering at different scales, i.e., views of clouds from close and far, and seamless transitions in between, requiring continuous LOD techniques;
- temporal cloud animations implying dynamic volumes and employing cloud simulation models.

ACKNOWLEDGEMENTS

Work supported by Austrian FFG Grant #830029.

6 REFERENCES

[1] D. Bernabei, F. Ganovelli, N. Pietroni, P. Cignoni, S. Pattanaik, and R. Scopigno. Real-time single scattering inside inhomogeneous materials. *The Visual Computer*, 26(6-8):583–593, 2010.

[2] A. Bouthors and F. Neyret. Modeling clouds shape. In *Eurographics, Short Presentations*, 2004.

[3] A. Bouthors, F. Neyret, and S. Lefebvre. Real-time realistic illumination and shading of stratiform clouds. In *EG Workshop on Natural Phenomena*, Vienna, Austria, 2006. Eurographics.

[4] A. Bouthors, F. Neyret, N. Max, É. Bruneton, and C. Crassin. Interactive multiple anisotropic scattering in clouds. In *Proc. ACM Symp. on Interactive 3D Graph. and Games*, 2008.

[5] E. Cerezo, F. Perez-Cazorla, X. Pueyo, F. Seron, and F. X. Sillion. A survey on participating media rendering techniques. *The Visual Computer*, 2005.

[6] D. Cha, S. Son, and I. Ihm. GPU-assisted high quality particle rendering. *Comp. Graph. Forum*, 28(4):1247–1255, 2009.

[7] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graph. and Games (I3D)*, Boston, MA, Etats-Unis, February 2009. ACM, ACM Press.

[8] C. Crassin, F. Neyret, M. Sainz, and E. Eisemann. *GPU Pro*, chapter X.3: Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels, pages 643–676. AK Peters, 2010.

[9] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann. Interactive indirect illumination using voxel cone tracing. *Comp. Graph. Forum (Proc. Pacific Graph. 2011)*, 30(7), 2011.

[10] Y. Dobashi, K. Kaneda, H. Yamashita, T. Okita, and T. Nishita. A simple, efficient method for realistic animation of clouds. In *Comp. Graphics (SIGGRAPH '00 Proc.)*, pages 19–28, 2000.

[11] D. S. Ebert, F. Musgrave, P. Peachey, K. Perlin, and S. Worley. *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann, 3rd edition, 2003.

[12] V. Forest, L. Barthe, and M. Paulin. Real-time hierarchical binary-scene voxelization. *Journal of Graphics, GPU, & Game Tools*, 14(3):21–34, 2009.

[13] E. Gobbetti, F. Marton, and J. A. Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806, 2008. Proc. CGI 2008.

[14] O. Gourmel, A. Pajot, M. Paulin, L. Barthe, and P. Poulin. Fitted BVH for fast raytracing of metaballs. *Comp. Graph. Forum*, 29(2):281–288, May 2010.

[15] M. J. Harris, W. V. Baxter, Th. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proc. SIGGRAPH/Eurographics '03 Conference on Graph. Hardware*, pages 92–101. Eurographics Association, 2003.

[16] M. J. Harris and A. Lastra. Real-time cloud rendering. *Comput. Graph. Forum (Proc. IEEE Eurographics '01)*, 20(3):76–84, 2001.

[17] K. Hegeman, M. Ashikhmin, and S. Premože. A lighting model for general participating media. In *Proc. Symp. on Interactive 3D Graph. and Games (I3D '05)*, pages 117–124, New York, NY, USA, 2005. ACM.

[18] R. Hufnagel, M. Held, and F. Schröder. Large-scale, realistic cloud visualization based on weather forecast data. In *Proc. IASTED Int. Conf. Comput. Graph. and Imaging (CGIM'07)*, pages 54–59, February 2007.

[19] W. Jarosz, C. Donner, M. Zwicker, and H. W. Jensen. Radiance caching for participating media. *ACM Trans. Graph.*, 27(1):1–11, 2008.

[20] W. Jarosz, D. Nowrouzezahrai, R. Thomas, P. P. Sloan, and M. Zwicker. Progressive photon beams. *ACM Trans. Graph. (SIGGRAPH Asia 2011)*, 30(6):181:1–181:12, December 2011.

[21] C. Knaus and M. Zwicker. Progressive photon mapping: A probabilistic approach. *ACM Trans. Graph.*, 30(3):25:1–25:13, May 2011.

[22] J. M. Kniss, S. Premože, Ch. D. Hansen, and D. S. Ebert. Interactive translucent volume rendering and procedural modeling. In *Proc. IEEE Visualization '02*, pages 109–116. IEEE Comput. Society Press, 2002.

[23] J. M. Kniss, S. Premože, Ch. D. Hansen, P. Shirley, and A. McPherson. A model for volume lighting and modeling. *IEEE Trans. Visualiz. Comput. Graph.*, 9(2):150–162, 2003.

[24] H.-J. Koppert, F. Schröder, E. Hergenröther, M. Lux, and A. Trembilski. 3d visualization in daily operation at the dwd. In *Proc. ECMWF Worksh. on Meteorolog. Operat. Syst.*, 1998.

[25] S. Laine and T. Karras. Efficient sparse voxel octrees.

In *Proc. ACM SIGGRAPH Symp.on Interactive 3D Graph.and Games (I3D '10)*, pages 55–63, New York, NY, USA, 2010. ACM.

[26] H. P. A. Lensch, M. Goesele, Ph. Bekaert, J. Kautz, M. A. Magnor, J. Lang, and H.-P. Seidel. Interactive rendering of translucent objects. In *Proc. Pacific Conf. on Comp. Graph. and Appl. (PG '02)*, pages 214–. IEEE Computer Society, 2002.

[27] N. Max, G. Schussman, R. Miyazaki, and K. Iwasaki. Diffusion and multiple anisotropic scattering for global illumination in clouds. *Journal of WSCG 2004*, 12(2):pp. 277, 2004.

[28] A. Miller, V. Jain, and J. L. Mundy. Real-time rendering and dynamic updating of 3-d volumetric data. In *Proc. Worksh. on General Purpose Process. on Graph. Process. Units (GPGPU-4)*, pages 8:1–8:8, New York, NY, USA, 2011. ACM.

[29] F. Neyret. A phenomenological shader for the rendering of cumulus clouds. Technical Report RR-3947, INRIA, May 2000.

[30] T. Nishita, Y. Dobashi, and E. Nakamae. Display of clouds taking into account multiple anisotropic scattering and skylight. *Comput. Graphics (SIGGRAPH '96 Proc.)*, pages 379–386, 1996.

[31] K. Perlin and E. M. Hoffert. Hypertexture. In *Comp. Graph. (SIGGRAPH '89 Proc.)*, volume 23 (3), pages 253–262, July 1989.

[32] S. D. Porumbescu, B. Budge, L. Feng, and K. I. Joy. Shell maps. *ACM Trans. Graph.*, 24:626–633, July 2005.

[33] S. Premože, M. Ashikhmin, R. Ramamoorthi, and Sh. K. Nayar. Practical rendering of multiple scattering effects in participating media. In *Proc. Eurographics Worksh. on Render. Tech.*, pages 363–373. Eurographics Association, June 2004.

[34] Z. Ren, R. Wang, J. Snyder, K. Zhou, X. Liu, B. Sun, P. P. Sloan, H. Bao, Q. Peng, and B. Guo. Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. In *ACM Trans. Graph. (SIGGRAPH '06 Proc.)*, pages 977–986, 2006.

[35] K. Riley, D. S. Ebert, Ch. D. Hansen, and J. Levit. Visually accurate multi-field weather visualization. In *Proc. IEEE Visualization (VIS'03)*, pages 37–, Washington, DC, USA, 2003. IEEE Computer Society.

[36] J. Schpok, J. Simons, D. S. Ebert, and Ch. D. Hansen. A real-time cloud modeling, rendering, and animation system. In *Proc. SIGGRAPH/Eurographics '03 Symp. Computer Anim.*, pages 160–166. Eurographics Association, July 2003.

[37] P. P. Sloan, N. K. Govindaraju, D. Nowrouzezahrai, and J. Snyder. Image-based proxy accumulation for real-time soft global illumination. In *Proc. Pacific Conf. on Comp. Graph. and Appl.*, pages 97–105, Washington, DC, USA, 2007. IEEE Computer Society.

[38] P. P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *ACM Trans. Graph. (SIGGRAPH '02 Proc.)*, pages 527–536, July 2002.

[39] L. Szirmay-Kalos, B. Tóth, and M Magdics. Free path sampling in high resolution inhomogeneous participating media. *Comp. Graph. Forum*, 30(1):85–97, 2011.

[40] S. Thiedemann, N. Henrich, Th. Grosch, and St. Mueller. Voxel-based global illumination. In *ACM Symp. on Interactive 3D Graph. and Games (I3D)*, 2011.

[41] A. Trembilski and A. Broßler. Surface-based efficient cloud visualisation for animation applications. *Journal of WSCG*, 10(1–3), 2002.

[42] N. Wang. Realistic and fast cloud rendering in computer games. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1, New York, NY, USA, 2003. ACM.

[43] Y. Yue, K. Iwasaki, B. Y. Chen, Y. Dobashi, and T. Nishita. Unbiased, adaptive stochastic sampling for rendering inhomogeneous participating media. *ACM Trans. Graph. (SIGGRAPH Asia 2010)*, 29:177:1–177:8, December 2010.

[44] K. Zhou, Z. Ren, St. Lin, H. Bao, B. Guo, and H. Y. Shum. Real-time smoke rendering using compensated ray marching. *ACM Trans. Graph.*, 27(3):36:1–36:12, August 2008.

Interactive Segmentation of Volume Data Using Watershed Hierarchies

Michal Hučko
Comenius University, Slovakia
michal.hucko@fmph.uniba.sk

Miloš Šrámek
Austrian Academy of Sciences, Austria
milos.sramek@oeaw.ac.at

ABSTRACT

The available methods for volume data segmentation and/or classification differ in the amount of the required user input on the one side and precision and ability to tweak the obtained results on the other. Automation of the task is more difficult when a general case is considered. In this paper we present an interactive segmentation and classification tool for arbitrary volumetric data, which is based on pre-segmentation of the volume in a hierarchy of homogeneous regions. The hierarchical subdivision allows for interactive adaptation of scale and precision according to the user requirements. The data is processed in three dimensions which minimises the amount of the needed interaction and gives instant overview of the resulting segmentation.

Keywords

Segmentation, user interface, watershed, scale-space.

1 INTRODUCTION

Segmentation methods, which are commonly used with volume data, can be classified in two groups – general methods and model-based methods [PB07]. Unlike general ones, model-based methods are based on certain knowledge about the target objects in the data as, for example, the expected object shape, mean voxel intensity, etc. In this paper we omit these as our goal is to provide a general segmentation tool which can be used with any volume data to segment arbitrary objects.

As the general methods use no additional information about the data which would aid in the process of segmentation, they require a greater amount of user interaction either in the form of process control, specification of parameters tailored to the current task or post-processing of the result. Our aim is to minimise this interaction while still leaving full control of the segmentation process to the user.

2 RELATED WORK

Common approach in volume data segmentation consists of selection of a region or object of interest in slices of the volume. The most basic general segmentation method used is manual segmentation where a user delineates the object of interest in the slices by hand. Although it is applicable at all times, its heavy user interaction demands are apparent. To speed up delineation of contours the LiveWire method [MMBU92] may be used. To obtain the desired results, a suitable cost function has to be first specified. The LiveWire method speeds up the process of contour drawing if the object of interest is clearly separeted from the rest of the data. If this condition is not satisfied for the current task, difficulties in cost function specification arise resulting in slow downs – user intervention is required and the method is reduced to manual segmentation.

As commonly used data sets have rather large dimensions, performing the segmentation on each slice is tedious and time consuming. An option is to segment only certain slices and interpolate the contour in the in-between slices [SPoP00]. Instead of the interpolation of the contour one may interpolate LiveWire control points instead and let the system compute contour in intermediate slices from the interpolated control points [SOB]. Precision of the resulting segmentation is dependent on the used interpolation and also on the set of key-slices. Problems might arise when topology or shape of the contour change rapidly between slices. Validation and potential correction of the interpolation is necessary.

Another common segmentation method is thresholding where voxels with intensities in certain range are selected. This method can be easily applied to certain data where tissue types can be distinguished by intensity (e.g. bone tissue in CT data), but applications to other data modalities or tissue types may pose a problem. If the task is to separate various objects of the same

tissue type thresholding may be used for preprocessing to reject clearly non-target voxels and thus localise the area of interest.

Two methods based on detection of homogeneous regions instead of contours are worth mentioning. General region growing requires user specified seed points and a suitable homogeneity criterion. Data is flooded from the seed points as long as the homogeneity criterion is satisfied, creating a homogeneous region. The watershed segmentation [VS91] is usually performed on a gradient image, which is treated as a topographic height map. Homogeneous regions with voxels with small gradient magnitude form valleys having voxels with high gradient magnitude on region borders as ridges. Although there exist various algorithms for computing the watershed transform, variants of two approaches are common – simulation of downhill water flow for each voxel or immersing of the relief into water. Exhaustive study of existing watershed algorithms is provided by Roedink and Meijster [RM00].

A broader overview and more detailed description of the existing segmentation methods can be found in [PB07].

2.1 Interaction techniques

All of the previously mentioned segmentation approaches differ in the way how a user can interact with the data to modify a partial result until the desired segmentation is achieved. For example, when delineating a contour the user directly sees the partial result and can undo recent steps if the contour starts to diverge from the desired position.

Thresholding requires numeric input – the threshold. Usually, a user is provided with a histogram from which the most suitable threshold value can be estimated. Clean separation of various tissue types based only on voxel intensity is rare and thus it may be difficult to find an optimal threshold value. Usage of other methods for refining the segmentation from thresholding (morphologic operations, connected component labeling, etc) is therefore convenient [STBH92].

Watershed segmentation produces a highly oversegmented result, especially if the data is spoiled by noise. Some methods allow merging of neighbouring regions if the shared border is weak (gradient magnitude is low). This situation is illustrated in figure 1 which shows gradient magnitude image of a CT head dataset slice – the corresponding watershed segmentation can be seen in figure 3 (red borders). The aim is to create segmentation in which the target object is labeled by a unique label. If this is not happening user intervention is usually required. For example, an interaction technique called marker-based watershed segmentation can be used [HP03], where a user specifies special include and exclude points in the data which prohibit merging of neighbouring regions if new region would contain markers of both types.

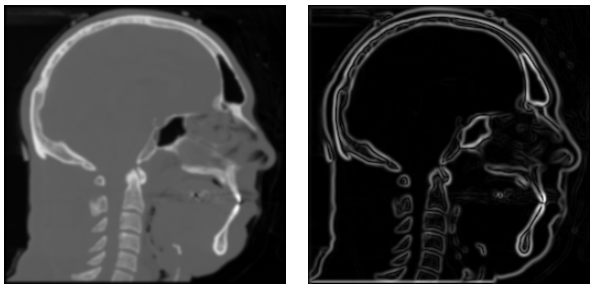


Figure 1: Slice from the Visible human male [Ack98] CT dataset with corresponding gradient magnitude image on right.

Watershed hierarchies [Beu94] were used in a technique based on interaction with slice views by Cates et al [CWJ05]. In such hierarchy, the order in which regions are merged defines a binary merge tree. Original regions form leaves and non-leaf nodes represent regions formed by merging of two regions – its children nodes. This tree may be used to segment an object from a data with a possibility to select large parts of the object by specification of high-positioned tree nodes and to refine the border by adding/removing low-positioned nodes.

In the paper by Armstrong et al [APB07] an extension of LiveWire or Intelligent Scissors called Live Surface was proposed. Data is presegmented into hierarchy of regions. Initial regions are computed using tobogganing [MB99] creating result equivalent to watershed segmentation (depends on the used watershed definition). For merging of the regions for higher levels in the hierarchy a special metric is used, which is based on the mean voxel intensity/colour and intensity/colour variance of a region. Segmentation is done by specifying two types of markers – inner and outer – which are then used to create a graph-cut in the region neighbourhood graph with minimal cost. Markers can be entered on an arbitrary cross section of the volume or directly in the 3D view allowing to add/remove parts to/from the border of an already segmented object.

3 THE SEGMENTATION TOOL

All of the segmentation approaches mentioned in the previous section were either proposed for two dimensional images or, if targeted to segmentation of 3D data, were used only for interaction in two dimensional space – on respective slices – or provided limited possibilities to modify the resulting segmentation directly in the 3D visualisation of the data. In our approach we let the user directly control the segmentation process by selecting fragments of the target object in 3D space. As manual segmentation by pixels/voxels is too cumbersome, data

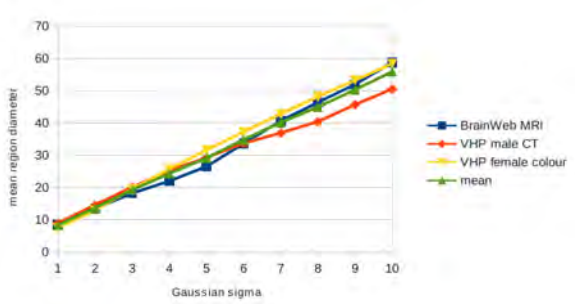


Figure 2: Measured dependency of mean region size on the Gaussian filter sigma. Three data sets were used – an MRI head, VHP male CT head and VHP female colour head dataset. Measured was diameter of minimal bounding sphere centered at the region's centre of mass.

is pre-segmented by the watershed transform as it creates homogeneous regions which can be used instead of single voxels.

3.1 Preprocessing

The watershed segmentation produces highly oversegmented results, because for each local minimum in the (usually gradient image of) original data a region is created. This can be a serious problem especially if noise is present in the data. To cope with this, data is usually first smoothed by the Gaussian filter. Smoothing, however, also shifts edges and removes weak edges, resulting in merging of regions which a user might desire in certain cases separated. Therefore, we perform the watershed segmentation on a sequence of derived data sets where each one is produced from the original by smoothing with a gradually increasing degree (increasing of the Gaussian sigma). As Koenderink showed [Koe84] this produces a set of images where on each image details smaller than certain size are increasingly ignored (the scale-space approach). As can be seen in figure 2, measurements on three different head datasets showed that dependency of mean bounding sphere diameter of regions on used Gaussian filter sigma can be approximated with function 1.

$$d(\sigma) = 5(\sigma - 1) + 10 \qquad (1)$$

To further reduce the starting number of regions, region merging based on mean region intensity or some other criterion can be used.

In order to correct position of the shifted edges, caused by smoothing, to the original position specified by the unsmoothed data or at least data at the lowest level of smoothing, an aligned region hierarchy is build. Spatial alignment of borders of corresponding regions on different hierarchy levels is achieved by the technique based on the maximum number of spatially overlapping voxels [SD02]. Thus, if later required, aligned regions

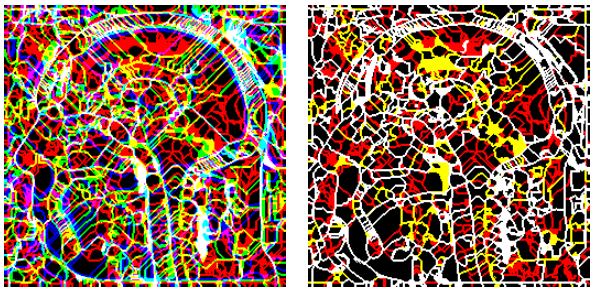


Figure 3: Three levels of watershed segmentation on visible human – male CT dataset. In the red channel are boundaries of regions for Gaussian smoothing with sigma 3, in green for sigma 5 and in blue for sigma 7. Overlapping boundaries are blended. On the left are the original watersheds, on the right the regions are aligned showing only red, yellow (red plus green) and white (all three levels) boundaries.

positioned higher in the hierarchy can be unambiguously decomposed to multiple smaller regions on some lower level.

As the last step of preprocessing the relevant neighbour information is extracted from the original voxel data and stored in a file. The described technique works equally well for scalar data (CT and MRI scans) as well as for multi-field data (dual energy CT, T1, T2 and PD MRI data etc). In the second case the watershed transform is performed on a gradient volume obtained as maximum of individual gradient fields.

3.2 GUI interaction

GUI of the segmentation application is shown in Figure 4. In this section all parts of the application as well as the segmentation workflow will be explained.

After the hierarchy is loaded into the application it is displayed in a tree widget allowing the user to navigate through it. Hovering or selecting a node highlights the corresponding region in a 3D visualisation window (Figure 5). This allows the user to choose an initial fragment of the region of interest. Depending on the target object and created hierarchy, regions on higher levels might consist of multiple target objects (e.g. various bones of skull). As pre-segmentation was performed at various scale levels, there is no need to repeat the process when the aforementioned problem arises – moving to lower levels in the hierarchy until the objects are separated is possible.

After the initial region is selected it is possible to display neighbouring regions which satisfy certain similarity criteria (Figure 6, also see section 3.3). The user can then select either one neighbour belonging to the target object by clicking into the 3D view or by selecting the neighbour in the list or can select all neighbors. Deselection of an undesired region is possible in a similar way, too. Once a neighbour is added/removed to/from

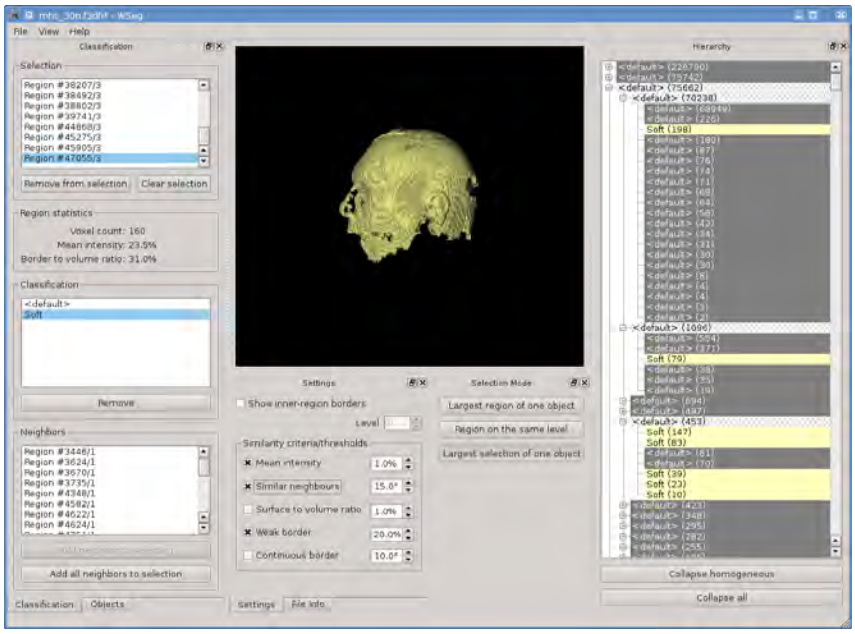


Figure 4: The segmentation tool: the region hierarchy is on the right. Left side from top to bottom: the list of selected regions, classification classes, the list similar neighbours of the regions in selection. At centre-bottom similarity criteria can be specified and above is the 3D visualisation and interaction window.

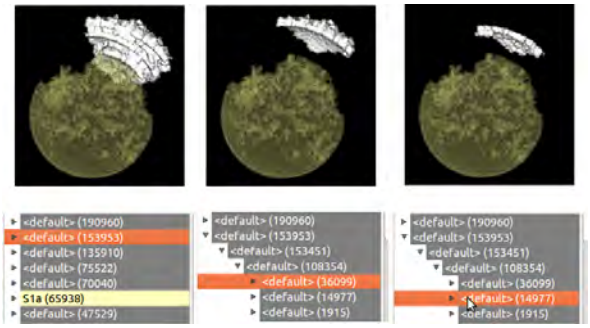


Figure 5: Selection of objects using the tree widget. The scene consist of concentric spheres/ellipsoids with different density and was intentionally spoiled by noise and superimposed low frequency density variation to make segmentation of the ellipsoids by thresholding impossible. Left: A region on the highest hierarchy level (white) was selected. One can see that it is composed of pieces of several ellipsoids. I we go down in the hierarchy (bottom row, in this case by two levels of the hierarchy), these pieces are separated and can be individually selected. The yellow region is a partially segmented and labeled part of the central sphere.

the selection, updated neighbours of the new selection are displayed. This allows traversal of the target object until all fragments are in the selection. Now classification class for the object can be created and all selected regions can be classified (Figure 6 right). Already classified regions are during selection refining displayed dimmed to provide context whereas the current selection is always highlighted.

3.3 Similarity criteria

To decide whether neighbouring regions are similar we implemented five different similarity criteria. Each criterion can be turned on or off and has a separate threshold value. When deciding whether two neighboring regions are similar all criteria which are turned on have to return a positive answer. Detailed description of the similarity criteria can be found below.

When searching for similar neighbours all neighbouring regions of the regions in the current selection are visited. As the selection might contain regions on different levels in the hierarchy we have to prevent multiple inclusion of same region – once directly and second time by including parent of the region. Naturally, regions positioned higher in the hierarchy are accepted in favor of the lower positioned child.

A second issue arises from the fact that we would like to move to a lower level of the hierarchy and to repeat the search for similar neighbors there, if we were not successful on a higher level. In this case, first, all regions in the selection are decomposed to regions on the lowest level – some lowest-level regions were directly present in the selection and some were selected indirectly by selection of its ancestor. Now we iterate through all pairs of the selected lowest-level regions and their neighbours (also at the lowest level). For both regions in the pair we ascend as many levels up in the hierarchy, until the original region in the selection which was decomposed to the currently processed lowest-level region is found. This leaves us with a list of pairs – (indirectly) selected region with its neighbour

– for each level not higher than level of the original selected region. Subsequently we start with the similarity tests. If a region and its neighbor on the higher level do not pass the selected similarity test we descent by one level and again perform the tests. By this higher positioned similar neighbours are found first.

A detailed description of the different similarity criteria follows:

Mean intensity Simple criterion comparing mean intensities of neighbouring regions. Difference in the intensities have to be in the specified interval for the test to pass. If the data contains multiple bands, the test must pass for all bands to be considered successful.

Similar neighbors In this criterion the prevalent directions of intensities in the region's neighborhood are compared. All neighbors of a region are visited and their center of mass is computed by averaging their geometric centroids weighted by their mean intensity. Subsequently, the direction vector, which points in the direction of growing intensity, is obtained by subtracting the center of mass from the geometric centroid of the region. Comparison of an angle between such vectors of two neighboring regions against a user defined threshold yields result of the similarity test.

Surface to volume ratio As working directly with the volume data would be slow and would increase memory requirements significantly, only derived information is used – mean intensity of a region, voxel count, etc. To compare region shapes ratio of the number of region's surface voxels to its total voxel count is used. The computed value is normalized to the $[0,1]$ range where 0 represents sphere-like objects and 1 string-like objects.

Weak borders For two regions to be similar in this criterion their border should have small mean gradient magnitude. Unlike the mean intensity criterion which compares mean values for whole regions, this criterion uses the original gradient data which were used during the creation of the most-detailed watershed segmentation. Intensity of the regions is n this criterion irrelevant.

Continuous border This criterion tries to find border in the data which spans multiple region boundaries. For two regions to pass this test they have to have a common neighbor. Both faces – first region with the common neighbor and second region with the common neighbor – have to be similar. For this, the angle between mean normal/gradient vectors of the faces is examined. Faces which are too small are ignored as their mean gradient vectors are based on too small set of values.

4 RESULTS

The application was implemented in C++ and uses OpenGL with GLSL. It was tested on the VHP dataset (the male head CT dataset). Individual bones of the skull, which are connected without a well defined border, were successfully segmented (figure 7 left). We also tested the application on an MRI scan of a human head. Figure 7 right shows segmentation of the brain cortex with certain gyri labeled as separate objects.

Preprocessing with three levels of watershed hierarchies and 3 additional levels of merging by density similarity took about 10 minutes. Both segmentations were produced in about 30 minutes.

In contrast to other methods, we intentionally omitted the possibility to interact through slices or cross sections to investigate the possibility for interaction only through the 3D view. If desired, display of the slices or cross sections can be easily added to the application, which may be then used for an initial selection.

If compared to traditional segmentation approaches, our method is most similar to region growing – assuming that mean intensity criterion is used. Neighbours to selected regions having similar intensity can be iteratively added to the selection until only regions with large difference in intensity remain. Because of the manual operation, user can omit regions which evaluate similar, but are not part of the target object. This can be essential when separating two or more objects of same or similar intensity, but different shape (as exemplified by segmentetion of the brain cortex in individual gyri). Automatic, or semi-automatic methods fail to separate these, if the interface is too weak or not present due to partial volume effect.

5 CONCLUSION

The presented segmentation and classification tool allows fast insight into data and fast segmentation of target structures while still leaving full control of the segmentation in user's hands. The amount of user interaction depends on data properties – resolution, presence of noise or other artefacts. Different similarity criteria were presented which should simplify and thus speed-up localisation of similar neighbouring regions in the data. Still, the user interaction is mainly done directly in the 3D visualisation window instead of slices. Operation in the visualisation window also gives direct visual feedback.

The concept was tested on the VHP dataset by segmentation of the bones of human skull and on MRI data by segmentation of respective gyri of the brain cortex.

6 ACKNOWLEDGEMENTS

This work was supported by the Slovak Research and Development Agency under the contract No. APVV-20-056105, Science Grant Agency (VEGA, Slovakia)

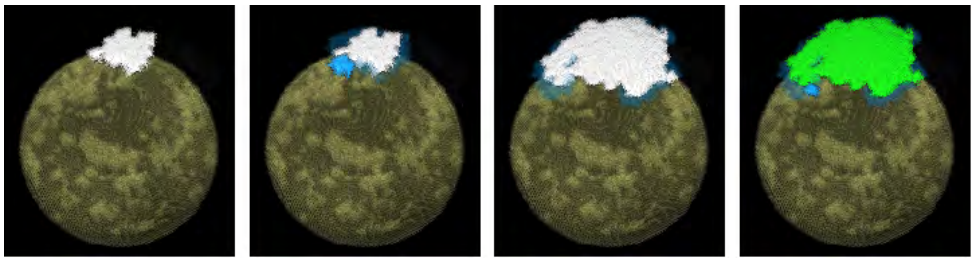


Figure 6: From left: selection of similar neighbors starting from an initial region. The blue regions are selection candidates (density similarity was used). White objects are already selected objects, the green color means that the object has already been labeled. The highlighted blue candidate object can be individually added to the selection or all candidates can be added at once.

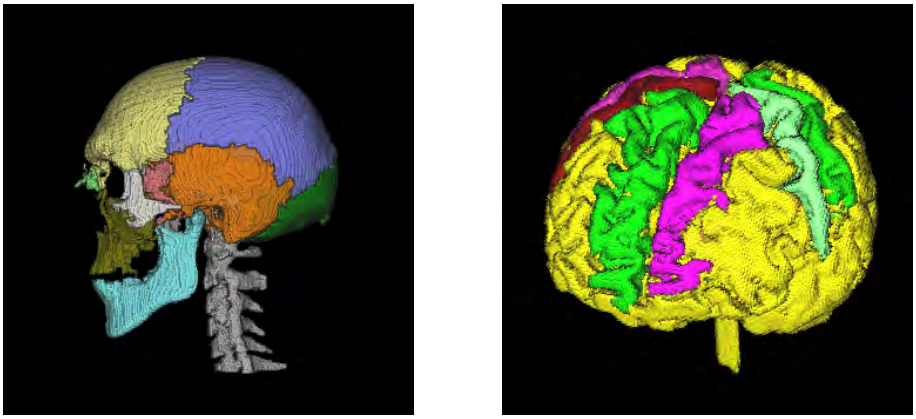


Figure 7: Left: segmentation of skull bones of the VHP male CT head dataset. Right: segmentation of brain and gyri in the MRI head dataset.

under contract No. 1/0631/11, FWF Austria under contract No. TRP-67-N23 and Comenius University under contract No. UK/229/2012. Miloš Šrámek is currently on leave at the the Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria.

7 REFERENCES

[Ack98] M.J. Ackerman, *The visible human project*, Proceedings of the IEEE **86** (1998), no. 3, 504–511.

[APB07] Christopher J. Armstrong, Brian L. Price, and William A. Barrett, *Interactive segmentation of image volumes with live surface*, Comput. Graph. **31** (2007), no. 2, 212–229.

[Beu94] S. Beucher, *Watershed, hierarchical segmentation and waterfall algorithm*, Mathematical morphology and its applications to image processing (J. Serra and P. Soille, eds.), Kluwer Academic Publishers, 1994, pp. 69–76.

[CWJ05] Joshua E. Cates, Ross T. Whitaker, and Greg M. Jones, *Case study: an evaluation of user-assisted hierarchical watershed segmentation.*, Medical Image Analysis (2005), 566–578.

[HP03] Horst K. Hahn and Heinz-Otto Peitgen, *Iwt-interactive watershed transform: a hierarchical method for efficient interactive and automated segmentation of multi-dimensional gray-scale images*, vol. 5032, SPIE, 2003, pp. 643–653.

[Koe84] J.J. Koenderink, *The structure of images*, Biological Cybernetics **50** (1984), no. 5, 363–370.

[MB99] E.N. Mortensen and W.A. Barrett, *Toboggan-based intelligent scissors with a four parameter edge model*, Proceedings of the IEEE Computer Vision and Pattern Recognition (CVPR '99), vol. II, 1999, pp. 452–458.

[MMBU92] Eric Mortensen, Bryan Morse, William Barrett, and Jayaram Udupa, *Adaptive boundary detection using Live-Wire two-dimensional dynamic programming*, IEEE Computers in Cardiology (1992), 635–638.

[PB07] B. Preim and D. Bartz, *Visualization in medicine: theory, algorithms, and applications*, The Morgan Kaufmann series in computer graphics, Morgan Kaufmann, 2007.

[RM00] Roerdink and Meijster, *The watershed transform: Definitions, algorithms and parallelization strategies*, FUNDINF: Fundamenta Informatica **41** (2000).

[SD02] M. Sramek and L. I. Dimitrov, *Segmentation of tomographic data by hierarchical watershed transform*, Journal of Medical Informatics and Technologies **3** (2002), 161–169.

[SOB] Zein Salah, Jasmina Orman, and Dirk Bartz, *Live-wire revisited*.

[SPoP00] Andrea Schenk, Guido Prause, and Heinz otto Peitgen, *Efficient semiautomatic segmentation of 3d objects in medical images*, In Proc. of Medical Image Computing and Computer-assisted Intervention (MICCAI, Springer, 2000, pp. 186–195.

[STBH92] T. Schiemann, U. Tiede, M. Bomans, and K. H. Höhne, *Interactive 3D-segmentation*, Visualization in Biomedical Computing 1992, Proc. SPIE 1808 (R. A. Robb, ed.), SPIE, Chapel Hill, NC, 1992, pp. 376–383.

[VS91] L. Vincent and P. Soille, *Watersheds in digital spaces: An efficient algorithm based on immersion simulations*, IEEE Transactions on Pattern Analysis and Machine Intelligence **13** (1991), 583–598.

Reconstructing Power Cables From LIDAR Data Using Eigenvector Streamlines of the Point Distribution Tensor Field

Marcel Ritter
Institute of Basic Sciences in Civil Engineering[2]
University of Innsbruck
marcel.ritter@uibk.ac.at

Werner Benger
Center for Computation & Technology[1]
Institute for Astro- and Particle Physics[2]
werner@cct.lsu.edu
werner.benger@uibk.ac.at

ABSTRACT

Starting from the computation of a covariance matrix of neighborhoods in a point cloud, streamlines are utilized to reconstruct lines of linearly distributed points following the major Eigenvector of the matrix. This technique is similar to fiber tracking in diffusion tensor imaging (DTI), but in contrast is done mesh-free. Different weighting functions for the computation of the matrix and for the interpolation of the vector in the point cloud have been implemented and compared on artificial test cases. A dataset stemming from light detect and ranging (LIDAR) surveying served as a testbed for parameter studies where, finally, a power cable was reconstructed.

Keywords: tensor-field visualization; streamlines; mesh-free methods; particle systems; point cloud; covariance matrix; fiber tracking; LIDAR; DT-MRI

1 INTRODUCTION

Reconstructing lines from point clouds has an important application in light detection and ranging applications (LIDAR). The surveying of power lines and their geometrical analysis is of great interest for companies that transmit electrical energy. Large networks of electric facilities have to be maintained to guarantee stable electrical power supply and prevent power outages. LIDAR surveying is a suitable technique to either detect damages on the electrical facilities or detect high growing vegetation in power line corridors [19] [15]. We experiment on a new method to reconstruct linear structures, stemming from airborne LIDAR surveying. We utilize a method inspired by diffusion tensor imaging (DTI) fiber tracking developed, originally, for magnetic resonance imaging (MRI) to track neuronal structures in the human brain [5].

1.1 Related Work

Current algorithms for reconstructing power lines are usually based on data filtering followed by a segmentation of the filtered and reduced point cloud either directly on the point cloud data or on a rastered 2D image. Melzer [18] first computes a digital terrain model (DTM) by using the method by Kraus [14] to remove terrain points. The remaining points are projected onto a 2D gray-scale raster (image). A Hough-Transform (e.g. [11]) is utilized iteratively to detect straight lines. Later, Melzer [17] improved the segmentation of LIDAR data also for power cables, based on the so called mean shift clustering, originally developed for pattern recognition [9]. Liu et al. [16] introduced a methodology based on statistical analysis to first remove ground points. Then, they project points onto a 2D gray-scale raster (image) and do a Hough-Transform similar to Melzer [18], but use a different technique for the Hough-Transform [8] to detect straight lines. Jwa et al. [13] developed a four step method. First they select power-line candidates, by utilizing a voxel based Hough-Transform to recognize linear regions. After a filtering process they construct line segments based on geometric orientation rules and, finally, use a voxel-based piece-wise line detector to reconstruct the line geometries.

Weinstein et al. [23] worked on tracking linear structures in diffusion tensor data stemming from MRI. Besides following the major Eigenvector they developed some rules for overcoming areas of not linear diffusion. The flow of Eigenvectors was also used for segmentation and clustering in brain regions as, for example, shown in [6] and [20]. Jones discusses the study of connections in human brains. He states that tracking the

[1] Louisiana State University, Baton Rouge, LA-70803, USA
[2] University of Innsbruck, Technikerstraße 13/4, A-6020 Innsbruck, Austria

diffusion directions is still not solved a in stable way and is an active research area [12].

Our work is based on previous work on the direct visualization of the covariance matrix describing the local geometric properties of a neighborhood distribution within in a point cloud, the so called point distribution tensor [21].

1.2 Our Approach

In our method we do not want to remove any points but operate on the entire dataset to avoid artifacts due to a complex point removal method. Instead, we first compute the point distribution tensor for each point. Eigen-analysis of the tensor yields the major Eigenvector, which indicates the dominant orientation of a point distribution. We may follow this orientation by computing streamlines along this dominant Eigenvector field in regions where one Eigenvalue dominates, so-called linear regions. In contrast, regions where the points are distributed more isotropic, are indicated by the point distribution tensor's Eigenvalues becoming more similar values. We want to avoid these regions, as they will not correspond to power cables. This approach is very similar to the fiber-tracking approach in medical visualization, but in our case the integration of the Eigenvectors needs to be done in a mesh-free way, merely on a point distribution rather than uniform grids. Thus, it can be applied to airborne LIDAR data without resampling to uniform grids (which would reduce data resolution and introduce artifacts due to the chosen resampling method).

1.3 Overview of the Paper

Section 2 presents the mathematical background and describes the implementation of the algorithm in section 2.2. Section 2.3 shows verifications by means of simple artificial point distributions. Here, the influence of different weighting functions on the tensor computation and the vector field interpolation during streamline integration is investigated. Also, two different numerical integration schemes are tested. In section 3 one set of power cables is reconstructed from a LIDAR data set stemming from actual observations. We then explore the available parameter space for weighting and integration in order to identify the best values for the given scenario.

2 ALGORITHM

2.1 Background

In [21] we defined the "point distribution tensor" of a set of N points $\{P_i : i = 1,...,N\}$ as

$$S(P_i) = \frac{1}{N}\sum_{k=1}^{N} \omega_n(|t_{ik}, r|)(t_{ik} \otimes t_{ik}^{\tau}), \qquad (1)$$

whereby $\otimes$ denotes the tensor product, $^{\tau}$ the transpose and $t_{ik} = P_i - P_k$. $\omega_n(|t_{ik}|, r)$ is a weighting function dependent on the distance of a point sample to a center point P_i and a radius of a neighborhood r, which can be constant or defined by a scalar field on the points: $r(P_i)$. We did not find a generally optimal solution for the weighting function, but implemented seven choices for our first investigations:

$$\omega_1 = 1 \qquad (2)$$

$$\omega_2 = 1 - x/r \qquad (3)$$

$$\omega_3 = 1 - (x/r)^2 \qquad (4)$$

$$\omega_4 = r/x^2 \qquad (5)$$

$$\omega_5 = \begin{cases} 1 - \frac{3}{2}a^2 + \frac{3}{4}a^3 & 0 \le a < 1 \\ \frac{1}{4}(2-a)^3 & 1 \le a < 2 \\ 0 & \text{otherwise} \end{cases} \qquad (6)$$

$$\omega_6 = \begin{cases} (\frac{5}{2} - b)^4 - 5(\frac{3}{2} - b)^3 + 10(\frac{1}{2} - v)^b & [0, \frac{1}{2}) \\ (\frac{5}{2} - b)^4 - 5(\frac{3}{2} - b)^3 & [\frac{1}{2}, \frac{3}{2}) \\ (\frac{5}{2} - b)^4 & [\frac{3}{2}, \frac{5}{2}) \\ 0 & [\frac{5}{2}, \infty) \end{cases} \qquad (7)$$

$$\omega_7 = \begin{cases} (3-c)^5 - 6(2-c)^5 + 15(1-c)^5 & [0,1) \\ (3-c)^5 - 6(2-c)^5 & [1,2) \\ (3-c)^5 & [2,3) \\ 0 & [3,\infty) \end{cases} \qquad (8)$$

with $a := \frac{2x}{r}$, $b := \frac{2.5x}{r}$ and $c := \frac{3.0x}{r}$, illustrated in Figure 1. The three functions ω_5, ω_6 and ω_7 are typical Gauss-like spline kernel functions used in smooth particle hydrodynamics (SPH) [10]. We use the same weighting functions for interpolating the vector field during Eigenvector integration. Even though, interpolation of Eigenvectors and interpolating tensors and locally computing its Eigenvectors lead to different results, we utilize the interpolation of the Eigenvector as a simpler implementation.

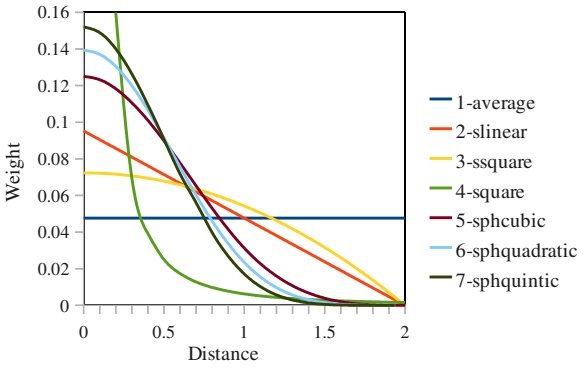


Figure 1: Different weighting functions of the distance interval 0.0 to 2.0, $r = 2.0$. Different slopes and characteristics are visualized. The *square* function (green) was clamped for axis scaling reasons and would grow further quadratically to the origin. The weights were normalized regarding to the integral of the curve in the interval. The curve numbers match the index of the weighting function: 1-average illustrates ω_1, 2-slinear illustrates ω_2, ...

We utilize tensor splats [1] for direct visualization of the tensor field. Figure 2 illustrates a point distribution along the edges of a rectangle and the corresponding tensor visualization with a neighborhood being $1/5$ of the longer rectangle edge. We then use Westin's shape

analysis method [24] to determine the so-called linear, planar and spherical shape factors. Points having a linearly distributed neighborhood are displayed as green oriented splats. Planar distributions are displayed as red disks. The linearity of the distribution tensor is shown in Figure 4 and Figure 5.

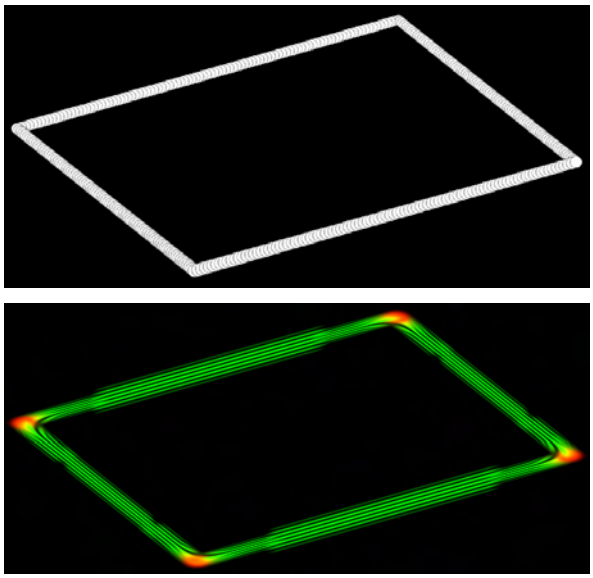


Figure 2: Distribution tensor visualization of a rectangular point distribution. *Top:* Points on a rectangle. *Bottom:* Tensor splats [1] of the point distribution tensor [21]. At each point one splat, a small textured and oriented disk, is drawn to represent the properties of the tensor's shape.

Visualizing streamlines is a common method to study vector fields. Starting from some seeding point, or initial condition, a curve $q(s)$ is computed which is always tangent to the vector field, solving the equation:

$$\dot{q}(s) = V(q(s)) \tag{9}$$

with s the curve parameter and V the vector field. Solving the differential equation at an arbitrary coordinate location Q within in the discretized data domain requires interpolation of the vector field. For mesh-free interpolation within a point cloud we use weighting functions parameterized with a specific radius of influence:

$$v(Q) = \frac{\sum_{i=1}^{N} v(P_i)\omega(|Q-P_i|,r)}{\sum_{i=1}^{N} \omega(|Q-P_i|,r)}, \tag{10}$$

with $v(P_i)$ representing the vector at point P_i.

2.2 Software Engineering Aspects

The algorithm was implemented using C++ within the VISH visualization shell [2]. The implementation extends a framework for computing integral geometries in vector fields, such as streamlines, pathlines or time surfaces. The streamline integration and visualization is separated into three different components: seeding, integration and displaying. The first component defines the initial conditions or seeding geometry. For computing streamlines within vector fields seeding points are sufficient. However, for streamlines of Eigenvector fields also an initial direction must be specified, because the Eigenvector is undirected. Integration based on an orientation continuing an user-chosen direction must be possible. Thus, requiring also a vector field on initial seeding points to disambiguate the Eigenvectors' orientations into unique directions.

Two new integration modules were developed. The first one extends the original streamline module, which was designed for vector field integration in uniform and curvilinear multi-block grids [4], to Eigenvector field integration. The second module expands this method further to allow integrating Eigenvector fields on mesh-free grids. One of the seven weighting functions (Equations 2, 3, 4, 5, 6, 7 and 8) and the radial influence weighting parameter can be specified for the interpolation of the Eigenvector inside the field domain. A range query on a KD-tree returns the points and their distances within the neighborhood of radius r. Equation 10 is utilized and Eigenvectors are aligned in orientation with respect to the Eigenvector of the closest neighbor. The Eigenvector is reversed if the dot product is negative. The integration of the streamline stops when the neighborhood becomes empty. Both integration modules support two different numeric schemes for the integration: explicit Euler and DOP853 [7]. Explicit Euler is used to get a fast yet inaccurate result. DOP853 is more expensive due to its adaptive stepsize but gives highly accurate results. When aiming at the same accuracy, DOP853 is faster than the Euler method by orders of magnitude. It is a Runge Kutta method of order eight using order five and three for error estimation and adaptive step size control, providing dense output. Accuracy measures and timing measures comparing the two integration methods were done, e.g., in [3].

The display module utilized here is reused from earlier development and implements color-coded illuminated lines utilizing OpenGL, allowing interactive navigation through the generated streamlines. Other modules, such as displaying ribbons [3] are also available.

2.3 Test Cases

We investigate the two Eigenvector integration modules on an uniform grid and on mesh-free grids. The Eigenvector field of a DTI-MRI scan [1], originally given on a uniform grid (128x128x56), was converted into a mesh-free grid, a point cloud holding the same Eigenvectors: Figure 3 (a) shows a volume rendering of the trace of the diffusion tensor along with the streamlines, revealing some brain structure and the location of the streamlines. Figure 3 (b) shows the comparison of Eigenvector streamlines computed on the uniform grid (blue) and Eigenvector streamlines computed in the point cloud (white). Both integrations were done with explicit Euler and a step size of 0.05. The size of a uniform grid cell is about 0.2, thus, utilizing about four integra-

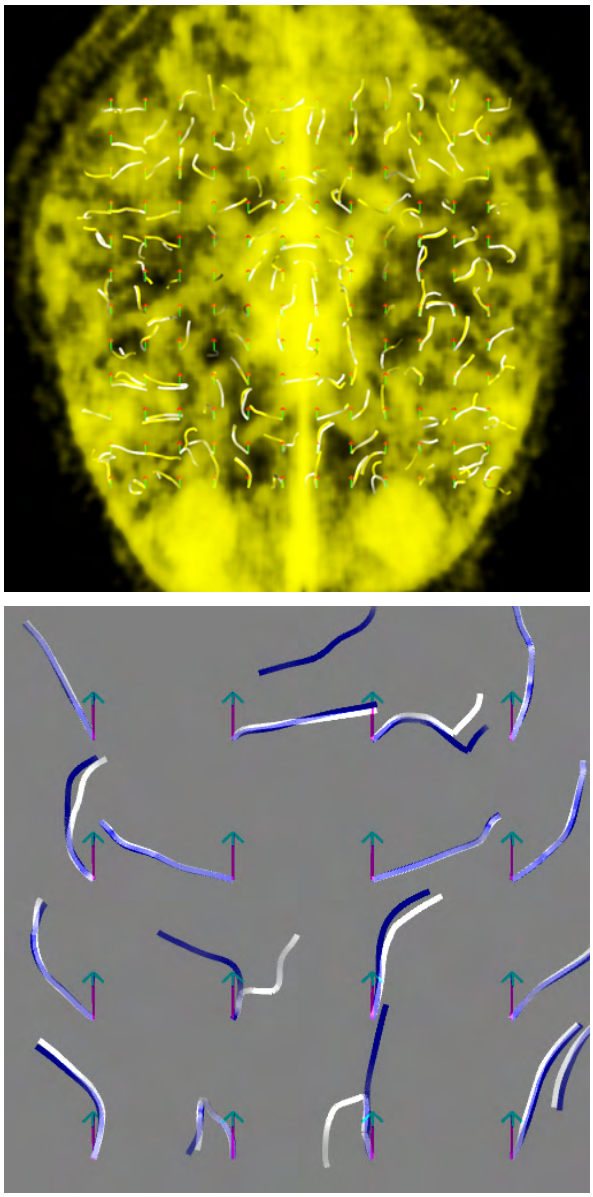
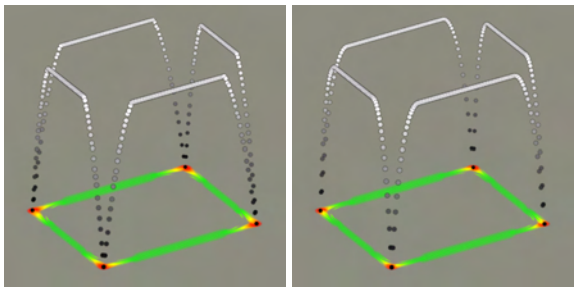
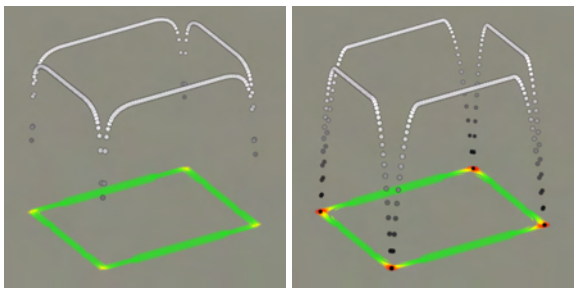


Figure 3: Comparison of the influence of integration of an Eigenvector field given on an uniform and a mesh-free grid. A mesh-free grid was generated from the uniform for testing. The arrows mark the start positions and directions of small Eigenstreamlines of a MRI diffusion tensor field. Streamlines on the uniform grid are blue. On the mesh-free grid they are white.

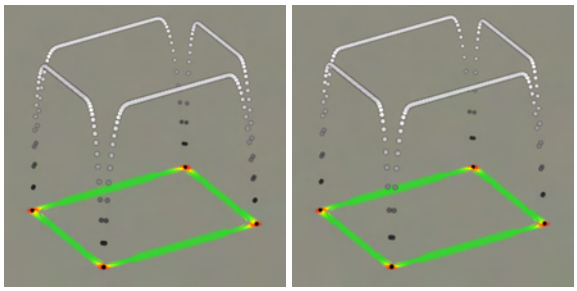
tion points per grid cell and requiring data interpolation within each cell. The length of each streamline is set to 1.0. Tri-linear interpolation was chosen for the uniform grid to compare the results with the linear weighting function ω_2 (*slinear*) for the mesh-free grid. The generated lines coincide on most cases. About 9% (13 of 144) do not coincide well. Some start in different directions. Here, the seeding vector field is almost perpendicular to the initial direction and the influence of the interpolation method results in different initial streamline directions. This issue could be cured by integrating Eigenvector streamlines in both directions starting from the initial seeding points, which would also allow avoiding the seeding vector field.



(a) average, slinear



(b) square, sphcubic

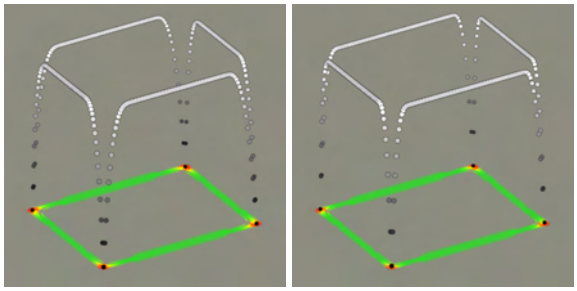


(c) sphquadratic, sphquintic

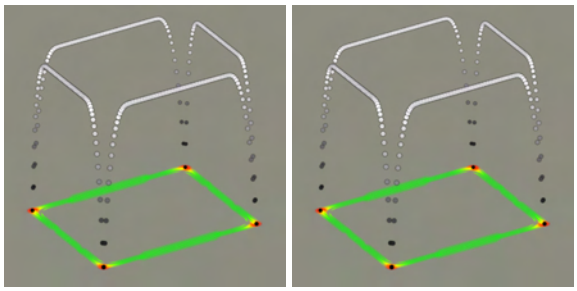
Figure 4: Influence of different weighting functions on the scalar field *linearity*, compare Figure 1. The linearity is illustrated by offset and over-scaling in z-axis, and gray-scale color-map on the points. Tensor splats directly show the distribution tensor.

Next, the influence of the different weighting functions on the computation of the distribution tensor was investigated. We define an analytic distribution of points along a rectangle as test case for computing the point distribution tensor. The rectangle is set up using a width of 10 and a height of 8. The radius parameter for the neighborhood is $r = 0.2$. Figure 4 illustrates the point distribution tensor using tensor splats and its corresponding linear shape factor by offsetting, over-scaling and a gray-scale color-map. The offsetting approach for the linear shape factor clearly illustrates the influence of the weighting: The "*average*" method resulting in a very abrupt change in the slope around corners points. The "*slinear*" weighting function results in smoother changes and a more localized influence, since closer points are weighted stronger than more distant points. *Square* shows the smoothest result. The three SPH spline kernels have an increasing locality with higher order of the kernel, when comparing *sphcubic*, *sphquadratic* and *sphquintic*. This is demonstrated in Figure 5 as well: Figure 5(a) shows the result of the

cubic and quadratic SPH kernel function. When the radius of the neighborhood is increased to match the kernels there is no visible difference between the *sphcubic* in Figure 5(a), *sphquadratic* and *sphquintic* in Figure 5(b) in the resulting *linearity*.



(a) *sphcubic* $r = 0.2$, *sphquadratic* $r = 0.2$



(b) *sphquadratic* $r = 0.25$, *sphquintic* $r = 0.3$

Figure 5: Different orders of the SPH kernel functions are compared, see Figure 1. *(a) sphcubic* and *shpquintic* using the same radius for the neighborhood. *(b) sphquadratic* and *shpquintic*, with adjusted neighborhood radius, have a similar result as the *sphcubic* (a)-left.

The influence of the integration scheme on the Eigenstreamline integration is demonstrated in Figure 6. The distribution tensor of a circular point distribution was computed using the *ssquare* weighting function. Tensor splats show the undirected Eigenvector, vector arrows show how the vector is directed within the internal vector representation. One Eigenstreamline is seeded downwards at the rightmost point of the circular point distribution and follows the undirected vectors. The top image shows Euler integration. Decreasing the step size would result in a more accurate integration. But, closing the gap of the integrated circle requires such a small step size, that the Runge Kutta method outperforms the Euler method. The 8^{th} order Runge Kutta method successfully closes the gap and reconstructs a circle from the circular point distribution, as shown in the bottom image. Also, a square-shaped point distribution was tested as shown in Figure 7. The length of a side is 10. Here, the influence of different weighting functions on the interpolation of the Eigenvector field was investigated. The distribution tensor was computed using the *ssquare* weighting function with $r = 2$. An Eigenstreamline is seeded downwards in the mid of the right edge. It follows the undirected vectors and flows around the corners of the rectangle. At each corner some error is introduced and the streamline is moving apart from

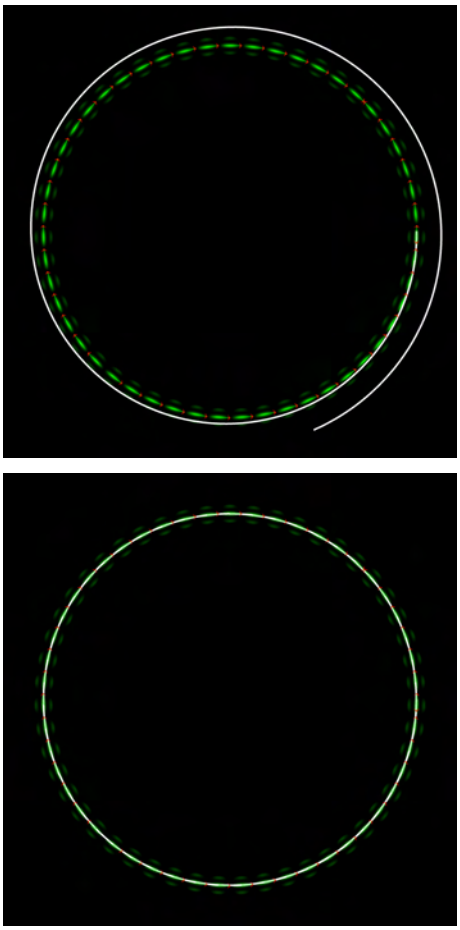


Figure 6: Comparison of different numerical integration schemes in a circular point distribution. One streamline (white) is seeded at the east-pole of the circle pointing southwards. Tensor splats and vector arrows illustrate the point distribution tensor and major Eigenvector. Note, that the Eigenvectors change orientation at north-east and south-west. *Top:* explicit Euler. *Bottom:* DOP853.

the original point distribution. Integration was done using the DOP853 method. Different weighting functions, mostly with $r = 1$, were tested for vector field interpolation. The length of the horizontal gap between the end and the start of the streamline was used as a measure for the integration error. Figure 8 shows the different errors in a bar diagram. The two best results were achieved using the *ssquare* and *average* weighting function.

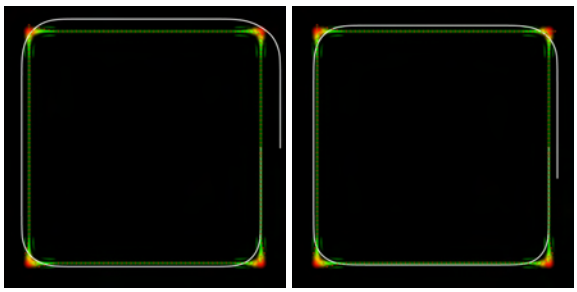


Figure 7: Comparison of Euler and DOP853 streamline integration on a square-shaped point distribution. Tensor splats and Eigenvectors are visualized besides the streamline (white) seeded downwards at the center of the right edge of the rectangle.

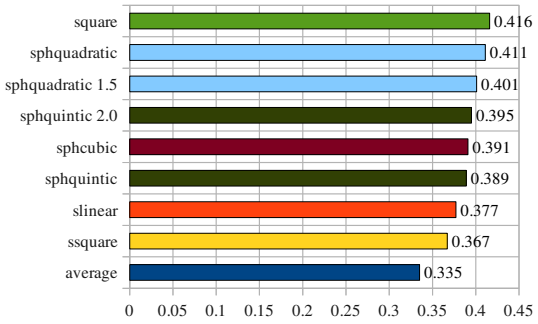


Figure 8: Comparison of errors in the square integration using different weighting functions for the vector interpolation. The weighting function for computing the tensor was *ssquare*, compare Figure 1. The values represent the horizontal distance between start and end point of the streamlines. The square's length is 10.0. The colors of the bars match the colors in Figure 1.

3 RESULTS

We used a dataset with circa eight million points covering a water basin close to the Danube in Austria. It was acquired by a Riegl's hydrographic laser scanner VQ-820G [22]. Figure 9 shows the point cloud colored by the *linearity* of a distribution tensor analysis. Here, we wanted to extract one power cable. The cable in the mid of the three lowest power cables suspended from the tall power pole. The white arrows mark the explicitly user-specified position and direction used as initial conditions of the streamline integration.

Different parameters and combinations of weighting functions for the tensor computation and the Eigenvector interpolation were investigated. The choice of a certain neighborhood radius and good weighting functions was crucial to successfully follow the 280 m long power cable. 41 parameter combinations were tested. For the tensor computation different radii $r = 0.5, r = 1.0$ and $r = 2.0$ and the weighting functions *average*, *slinear*, *ssquare* and the SPH kernels for the tensor were used. For the vector interpolation radii $r = 0.25, r = 0.5, r = 1.0, r = 2.0$ and $r = 3.0$ and all seven weighting functions were used. Figure 10 shows a view along the power cable, with a non optimal configuration. The Eigenstreamline is not following the cable to the end because it moves apart more than 1.0 m from the cable, resulting in an empty neighborhood during integration.

Best results were achieved by using the *ssquare* weighting with $r = 2.0$ for tensor computation and the *sphquintic* weighting with $r = 3.0$ for the vector interpolation. Results show that a more smooth weighting in the tensor computation and a more local interpolation weight are a good combination for reconstructing linear structures. Using the same weighting for tensor computation and vector interpolation did not work, see Figure 11 (b). The global error of the reconstruction at the end of the power cable is about 80 cm and needs to be further improved. The cable could only be followed using DOP853 integration. Explicit Euler failed to produce acceptable results. When comparing Figures $11(a)$ and $11(c)$ the global error is almost the same. The main difference is the local shape of the Eigenstreamline. A larger vector interpolation radius results in a smoother curve. Figure $11(c)$ shows the best reconstruction of the investigated technique and described parameters.

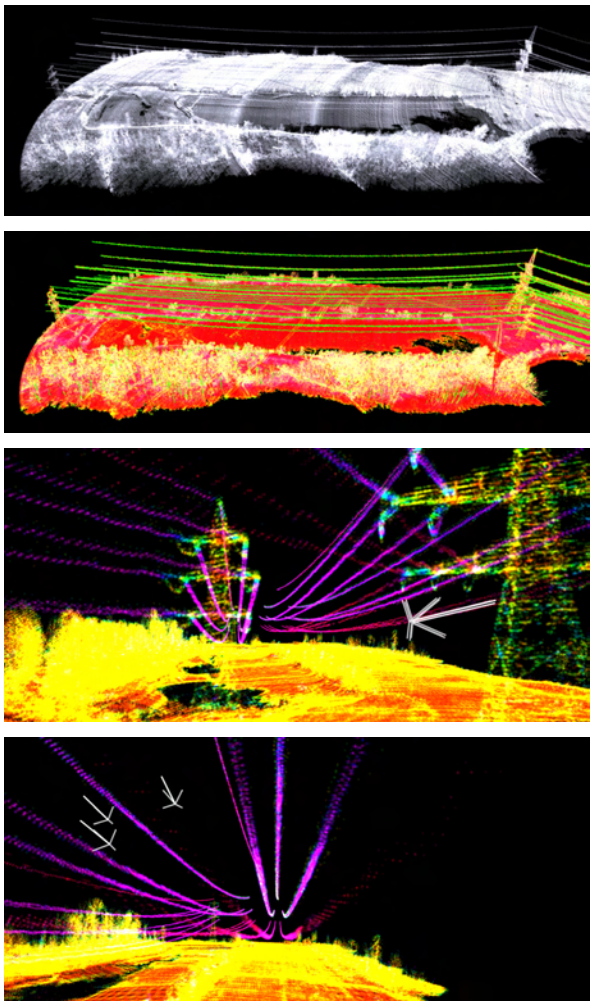


Figure 9: Overview of the LIDAR data set. The two upper images show the point cloud as points and as tensor splats (taken from [21]). In the two lower images points are colored by *linearity*. Three arrows mark the explicitly user-specified seeding points and directions of the streamline computation located at the mid lower power cable (magenta) of the larger power pole.

4 CONCLUSION

A new method of reconstructing power cables, or other linear structures in general, in point clouds was presented. The method employs the point distribution tensor as presented in previous work [21]. Different weighting functions for the tensor computation and the interpolation of the major Eigenvector field were implemented and compared. Streamline integration was verified on artificial test cases and applied to a LIDAR point cloud dataset acquired from actual observations. Finally, a power cable was reconstructed and visualized using this dataset.

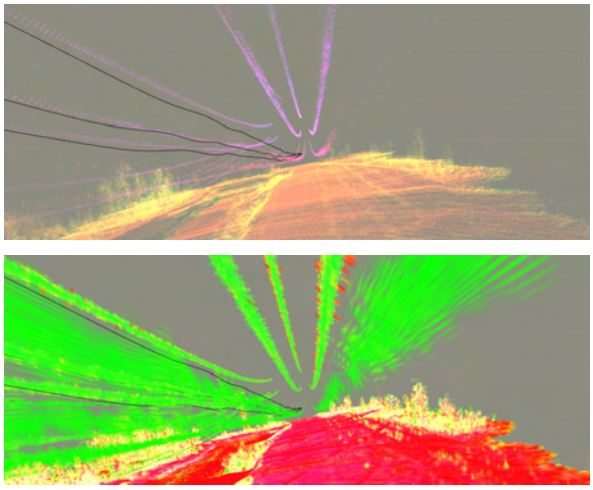
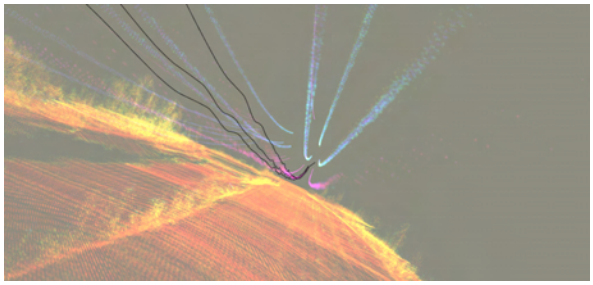
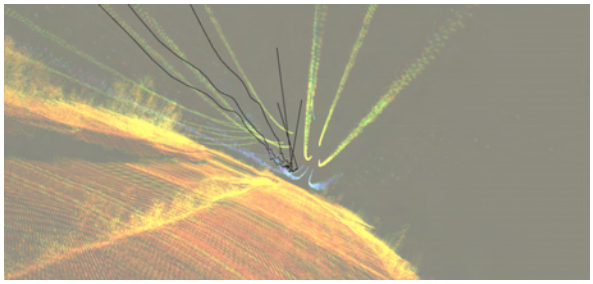


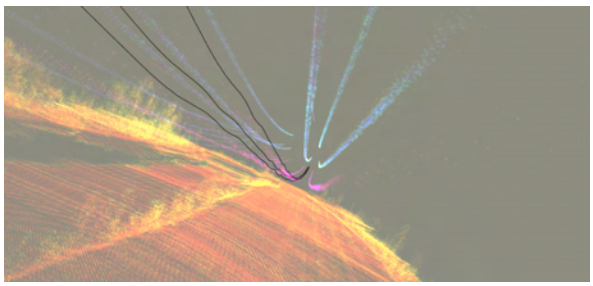
Figure 10: Power cable reconstruction via streamlines. The distribution tensor was computed using the *average* $r = 2.0$ weighting and the vector interpolation was done with the *ssquare* $r = 1.0$ weighting. *Top:* Points colored by *linearity*. *Bottom:* Tensor splats illustrate the distribution tensor. Streamlines are moving apart from the power cable and break before they can reconstruct the full 280 m of cable.



(a) Tensor: *ssquare r=2*, Vectorfield: *sphquintic r=1*



(b) Tensor: *sphquintic r=2*, Vectorfield: *sphquintic r=2*



(c) Tensor: *ssquare r=2*, Vectorfield: *sphquintic r=3*

Figure 11: Comparison of different parameters and weighting function combinations of the computation, finally resulted in a successfully reconstructed power cable. The LIDAR point cloud is colored by *linearity* of the distribution tensor. The three Eigenvector streamlines reconstruct a 280 m long cable.

5 FUTURE WORK

Other weighting functions for computing the tensor and doing the interpolation during the streamline integration need to be tested. Automatic determination of the optimal combination of weighting functions and also their parameters will be the goal of further investigations. Seeding points and directions for computing the streamlines need also to be chosen automatically, for example, by taking tensor properties into account. Following the major Eigenvector of points with high *planarity* or *sphericity* needs to be prevented during streamline integration. Finally, more datasets should be explored to stabilize the method. Furthermore, minor changes of the algorithm would enable streamline integration in datasets stemming from SPH simulations.

ACKNOWLEDGMENT

Many thanks to Frank Steinbacher for proving the LIDAR data. This work was supported by the Austrian Science Foundation FWF DK+ project *Computational Interdisciplinary Modeling* (W1227) and grant P19300. This research employed resources of the Center for Computation and Technology at Louisiana State University, which is supported by funding from the Louisiana legislature's Information Technology Initiative. This work was supported by the Austrian Ministry of Science BMWF as part of the UniInfrastrukturprogramm of the Forschungsplattform Scientific Computing at LFU Innsbruck.

REFERENCES

[1] W. Benger, H. Bartsch, H.-C. Hege, H. Kitzler, A. Shumilina, and A. Werner. Visualizing Neuronal Structures in the Human Brain via Diffusion Tensor MRI. *International Journal of Neuroscience*, 116(4):pp. 461–514, 2006.

[2] W. Benger, G. Ritter, and R. Heinzl. The Concepts of VISH. In 4^{th} *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007*, pages 26–39. Berlin, Lehmanns Media-LOB.de, 2007.

[3] W. Benger and M. Ritter. Using geometric algebra for visualizing integral curves. In E. M. S. Hitzer and V. Skala, editors, *GraVisMa 2010 - Computer Graphics, Vision and Mathematics for Scientific Computing*. Union Agency - Science Press, 2010.

[4] W. Benger, M. Ritter, S. Acharya, S. Roy, and F. Jijao. Fiberbundle-based visualization of a stir tank fluid. In 17^{th} *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 117–124, 2009.

[5] T. E. Conturo, N. F. Lori, T. S. Cull, E. Akbudak, A. Z. Snyder, J. S. Shimony, R. C. Mckinstry, H. Burton, and M. E. Raichle. Tracking neuronal

fiber pathways in the living human brain. *Proc Natl Acad Sci U S A*, 96(18):10422–10427, Aug. 1999.

[6] M. Descoteaux, L. Collins, and K. Siddiqi. A multi-scale geometric flow for segmenting vasculature in mri. In *In Medical Imaging Computing and Computer-Assisted Intervention*, pages 500–507, 2004.

[7] S. N. E. Hairer and G. Wanner. *Solving ordinary differential equations I, nonstiff problems, 2nd edition*. Springer Series in Computational Mathematics, Springer-Verlag, 1993.

[8] L. A. Fernandes and M. M. Oliveira. Real-time line detection through an improved hough transform voting scheme. *Pattern Recognition*, 41(1):299 – 314, 2008.

[9] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *Information Theory, IEEE Transactions on*, 21(1):32 – 40, jan 1975.

[10] D. A. Fulk and D. W. Quinn. An analysis of 1D smoothed particle hydrodynamics kernels. *Journal of Computational Physics*, 126(1):165–180, 1996.

[11] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2001.

[12] D. K. Jones. Studying connections in the living human brain with diffusion mri. *Cortex*, 44(8):936 – 952, 2008.

[13] Y. Jwa, G. Sohn, and H. B. Kim. Automatic 3d powerline reconstruction using airborne lidar data. *IAPRS*, XXXVIII(2004):105–110, 2009.

[14] K. Kraus and N. Pfeifer. Determination of terrain models in wooded areas with airborne laser scanner data. *ISPRS Journal of Photogrammetry and Remote Sensing*, 53(4):193 – 203, 1998.

[15] Z. Li, R. Walker, R. Hayward, and L. Mejias. Advances in vegetation management for power line corridor monitoring using aerial remote sensing techniques. In *Applied Robotics for the Power Industry (CARPI), 2010 1st International Conference on*, pages 1 –6, oct. 2010.

[16] Y. Liu, Z. Li, R. F. Hayward, R. A. Walker, and H. Jin. Classification of airborne lidar intensity data using statistical analysis and hough transform with application to power line corridors. In *Digital Image Computing : Techniques and Applications Conference (DICTA 2009)*, Melbourne, Victoria, December 2009. IEEE Computer Society.

[17] T. Melzer. Non-parametric segmentation of als point clouds using mean shift. *Journal of Applied Geodesy*, 1(3):159–170, 2007.

[18] T. Melzer and C. Briese. Extraction and modeling of power lines from als point clouds. In *Proceedings of 28th Workshop*, pages 47–54. Österreichische Computer Gesellschaft, 2004. talk: Austrian Association for Pattern Recognition (ÖAGM), Hagenberg; 2004-06-17 – 2004-06-18.

[19] S. Mills, M. Gerardo, Z. Li, J. Cai, R. F. Hayward, L. Mejias, and R. A. Walker. Evaluation of aerial remote sensing techniques for vegetation management in power line corridors. *IEEE Transactions on Geoscience and Remote Sensing*, October 2009.

[20] M. Persson, J. Solem, K. Markenroth, J. Svensson, and A. Heyden. Phase contrast mri segmentation using velocity and intensity. In R. Kimmel, N. Sochen, and J. Weickert, editors, *Scale Space and PDE Methods in Computer Vision*, volume 3459 of *Lecture Notes in Computer Science*, pages 119–130. Springer Berlin - Heidelberg, 2005.

[21] M. Ritter, W. Benger, B. Cosenza, K. Pullman, H. Moritsch, and W. Leimer. Visual data mining using the point distribution tensor. In *IARIS Workshop on Computer Vision and Computer Graphics - VisGra 2012*, Feb-Mar 2012.

[22] F. Steinbacher, M. Pfennigbauer, A. Ulrich, and M. Aufleger. Vermessung der Gewassersohle - aus der Luft - durch das Wasser. In *Wasserbau in Bewegung ... von der Statik zur Dynamik. Beitrage zum 15. Gemeinschaftssymposium der Wasserbau Institute TU München, TU Graz und ETH Zürich*, 2010.

[23] D. Weinstein, G. Kindlmann, and E. Lundberg. Tensorlines: advection-diffusion based propagation through diffusion tensor fields. In *Proceedings of the conference on Visualization '99: celebrating ten years*, VIS '99, pages 249–253, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[24] C. Westin, S. Peled, H. Gudbjartsson, R. Kikinis, and F. Jolesz. Geometrical diffusion measures for mri from tensor basis analysis. In *Proceedings of ISMRM, Fifth Meeting, Vancouver, Canada*, page 1742, Apr. 1997.

Real-time Mesh Extraction of Dynamic Volume Data Using GPU

Szymon Engel
AGH, Krakow, Poland
szymon@hoopoe.com.pl

Witold Alda
AGH, Krakow, Poland
alda@agh.edu.pl

Krzysztof Boryczko
AGH, Krakow, Poland
boryczko@agh.edu.pl

ABSTRACT

In the paper an algorithm of triangle mesh generation for a full three-dimensional volumetric data is presented. Calculations are performed in real time using graphics processors. The method is very well suited for the visualization of dynamic data, as the calculations use only the current frame data (not including data from previous frames). Due to high performance of the algorithm, it can be practically applied in programs for digital sculpting, simulators and games which require editable geometries.

Keywords

mesh generation, iso-surface, volume data, real-time, GPU computation

1 INTRODUCTION

Nowadays, visualization of volumetric data is very often applied in practice. Both in medicine, e.g. for the MRI, PET, or CT data presentation in medical imaging, nondestructive inspection of materials (industrial CT), digital sculpting software, as well as in computer games. Often the visualization itself is not sufficient and a three-dimensional mesh is required for a physical calculations, such as collision detection, calculation of material properties or stress. Moreover, the advantage of representing models with triangle meshes is that modern GPUs are optimized for efficient rendering of triangles.

Another issue is that volumetric data can change dynamically. When modeling is performed in a program for sculpting a virtual material, a three-dimensional mesh generated in real time is needed in order to display the results. In video games or simulators of earth-moving machineries, we have to deal with the terrain, which cannot be fully represented by height maps. We may require a visualization of structures such as tunnels, caves, overhangs of the land as well as other modifications caused as a result of a player actions, such as explosions, vehicle interactions with the terrain or other gameplay activities.

Currently available methods often do not allow to generate a mesh for a large amount of data in real time. Sometimes the resulting effect is described as "interactive" which usually means ability to carry out calculations giving a few frames per second. This speed, however, is not sufficient for the smooth operation of applications such as computer games.

In response to these problems, a mesh generation algorithm for a fully three-dimensional volumetric data has been developed. All calculations are performed on GPU in real time by which we understand the actual mean speed of computations above 30 frames per second. Another advantage of the presented algorithm is its independence of the volumetric data representation; therefore, scalar fields, implicit functions, or metaball objects can be used.

The rest of this article is organized as follows. The next section presents previous work on mesh generation methods of volumetric data. The third part briefly describes the possible methods of data representation. It then shows the subsequent steps of the algorithm while in the following passage detailed information on how to implement the algorithm using graphics processors is included. The last section presents a description and summary of results.

2 RELATED WORK

The subject of this paper is to visualize an iso-surface generated for the volumetric data using triangle mesh. Mesh-based methods allow their easy integration with other algorithms or modules, e.g. physics engines, that require mesh as an input. Thanks to this approach our algorithm can be used in real-time applications.

Therefore, discussion of previous works does not include ray-tracing and view-dependent visualization; however, information on the methods of these types can be found e.g. in [OBA05, LK10, GDL+02].

GPU-based methods which allows to interactive or real-time calculations were presented e.g. in [KOKK06, CNLE09, KOR08, TSD07].

One of the basic methods of generating a mesh on the basis of volumetric data is the Marching Cubes algorithm [LC87] developed in the 1980s. It consists in dividing the visualized space into equal cubes and generating a polygon within each cube, which is then subjected to triangulation. The position and shape of the polygon are dependent on the values of an implicit function in eight vertices of each cube. A common issue with this method is that it requires generating a mesh in the entire space of visualized data. In response to this, several hierarchical and adaptive versions of the Marching Cubes algorithm have been developed [OR97, Blo88, WKE99, KKDH07] using octal trees to reduce the area for which calculations were carried out and which reduce the number of triangles by generating them in different densities depending on the distance of the camera. Implementations of of the Marching Cubes algorithm using GPUs are presented in [JC06, Gei07].

Another, possibly less popular, method is the SurfaceNets method [Gib98] which originally was used to visualize binary medical data. This method is dual to the marching cubes algorithm and, as in the latter one, visualized space is divided into cubes of the same dimensions. In its base version, the method consisted of selecting the nodes belonging to the surface in a way that a certain node has been selected, if among the eight vertices of the cube were those that had a different sign. These nodes were linked with adjacent ones thus creating a net, which was then smoothed by moving the nodes so as to minimize the energy between them while maintaining the restriction that a node could not leave the cube, to which it originally belonged. The final step was a triangulation of the network, which gave the resulting mesh of triangles. The next method which belongs to the group of "dual algorithms" is the one by [Nie04], which generates a mesh very similar to SurfaceNets, but its implementation is more like the Marching Cubes algorithm. One of the important differences between this method and the SurfaceNets is that the mesh generated by the former is a proper two-dimensional manifold.

In addition to the methods outlined above, there are also: the marching tetrahedra method [TPG99], whose operating principle is based on the marching cubes algorithm, the marching triangles method based on the Delaunay triangulation [HSIW96] which generates an irregular mesh, or the method based on marching cubes which allows one to obtain sharp edges, described in the work [JLSW02].

We follow the existing approach of dual marching cubes, however, our algorithm is implemented exclusively on GPU and it efficiently exploits geometry shaders. Thanks to the use of dual methods, the resulting mesh contains fewer triangles and is regular due to the number of generated triangles within each of the cubes. The latter allows the method to be implemented in a very efficient way using graphics processors. Former GPU-accelerated mesh extraction algorithms (e.g. [KW05], [Goe05], [JC06]) are based on both CPU and GPU computations, using vertex or fragment shaders only. Although meshes generated using our method are not proper two-dimensional manifolds, our approach is extremely efficient and can be used for dynamic data which change on random every frame.

3 VOLUMETRIC DATA

The basic method of describing three-dimensional volumetric data is the implicit function. By setting the values of the contour we obtain surface limiting the desired area. If the values of this function represent the Euclidean distances from a given contour and we save them as an array, then we get a three dimensional signed distance field $D : \mathbb{R}^3 \to \mathbb{R}$, representing the iso-surface S, defined for point $p \in \mathbb{R}^3$ as:

$$D(p) = sgn(p) \cdot \min\{|p-q| : q \in S\}, \qquad (1)$$

where

$$sgn(p) = \begin{cases} -1 & \text{if p is inside} \\ +1 & \text{if p is outside} \end{cases} \qquad (2)$$

This representation can be stored in graphics card memory as a three-dimensional texture. Thanks to this representation, smoothing of the resulting mesh using normals computed directly from distance fields and vertex distances from the surface, is very effective.

Also, most medical data is stored in the form of three-dimensional arrays. For such data combined with contour values we can generate a mesh. Another, less common way to represent the volumetric data, is using metaball objects which, with adequate representation, can be converted to distance fields.

4 ALGORITHM OVERVIEW

Due to the GPU architecture and the way they carry out calculations, the developed algorithm is based on dual methods. They allow to obtain a regular mesh consisting of squares, so one doesn't need expensive triangulation of polygons generated inside a cube, as is the case of marching cubes method. The algorithm has been adapted to carry out calculations on GPUs, and highly parallelized, which allows it to achieve very high performance.

Input data block, of size n^3, where n of a form 2^k, is divided into equal cubes. This is shown in Figure 1.

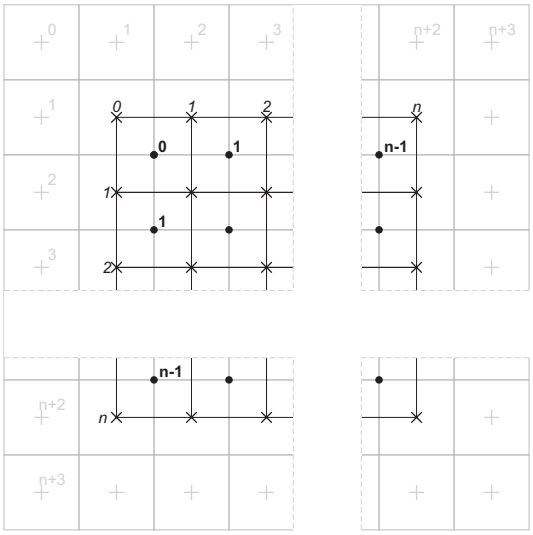


Figure 1: Data organization (2D view)

The image represents a two-dimensional version, but the three-dimensional one is similar.

Calculation points marked with "•" symbol are placed in the centers of cubes and their coordinates are used to calculate the distance from the contour, based on the volumetric data. Distance field values are stored in points marked "+", while points marked "×" represent vertices of quadrangles generated by the algorithm.

The calculations of the algorithm are processed in two steps. At the beginning, for each calculation point $p \in \mathbb{R}^3$, marked with the symbol "•", such that $p \in \{[0,n-1] \times [0,n-1] \times [0,n-1]\}$, there are generated three points $q,r,s \in \mathbb{R}^3$ such that for $p = (x,y,z)$: $q = (x-1,y,z)$, $r = (x,y-1,z)$, $s = (x,y,z-1)$. In each of these points p,q,r,s an implicit function value $d_p = f(p)$ is calculated and three edges defined by pairs of calculation points pq, pr, ps are created. Then, for each edge – if its endpoints have different signs (lie on different sides of the contour) – a quadrangle located on the border of the cubes is generated. Its orientation is determined by the direction of the edge, which is consistent with the direction of the normal to the surface of the quadrangle. A set of squares, generated in this way, approximates the iso-surface for input volumetric data, and is smoothed in the next stage. As a result of conversion of each square to a pair of triangles, a triangle mesh is obtained.

Due to the fact that during the calculation the sign changes, zero is treated differently depending on the direction of the edge, the condition 3, under which quadrilaterals are generated is presented as follows:

$$(f(q) \geq 0 \wedge f(p) < 0) \vee (f(q) < 0 \wedge f(p) \geq 0) \quad (3)$$

The first step in the algorithm, described above, is ideally suited for parallelization, because the calculations for computing the individual points can be carried out independently. Despite the fact that the distance fields are calculated for each point twice, it is possible to obtain a high-performance computing algorithm, because it is not a costly operation.

The second stage of the algorithm is a smoothing of generated mesh by moving its vertices in the direction of the surface represented by the distance fields values. For this purpose, at each vertex of the distance field, a normal vector n is calculated on the basis of the gradient and the distance d to the surface. Then, the vertex is moved in the direction of the normal, by the value calculated according to formula 4.

$$p' = p + dn \quad (4)$$

In the case that the resulting mesh is used only for displaying a surface, this step can be implemented directly during rendering. Otherwise, it is possible to smooth the mesh only and use it for subsequent calculations, such as collision detection.

The advantage of the developed algorithm over marching cubes method consists partly in that for creating a quad for the cube we generate exactly four indices and there is no need for triangulation of polygons generated. This allows the calculations to be successfully performed on the GPU.

5 IMPLEMENTATION DETAILS

The algorithm was implemented using the OpenGL graphics library; however, DirectX library can be used as well.

Input volumetric data, for which a mesh is generated, is divided into equal blocks, for which fragments of the mesh are generated independently. This approach was chosen because of the GPU hardware limitations on the maximum size of supported textures, as well as for optimization purposes. To be specific: not all parts of dynamic data need to be modified at the same time, and hence there is no need for mesh regeneration in these blocks. Generated meshes merge together in continuous blocks along borders.

The algorithm is carried out equally for each data block. All calculations are done on the GPU, so the data are not transferred between main memory and graphics card memory. Calculations are carried out in two phases, the algorithm flowchart is shown in Figure 2.

In both passes of the rendering shaders refer to the volumetric data.

5.1 Volume Representation

Three-dimensional floating-point texture is used for distance field representation in each block of data.

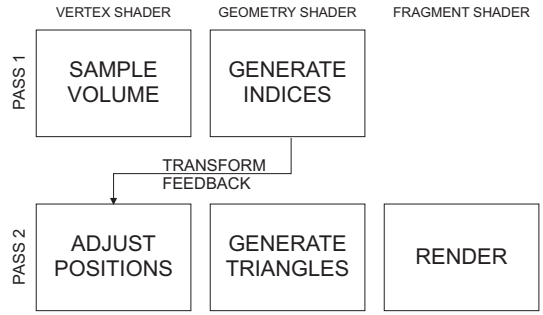


Figure 2: Algorithm flowchart

Texture size is $(n+3)^3$, 2 bytes per texel. Although 4-byte floating point numbers can be applied for the precision improvement, no need for that has been found during testing. To avoid communication between neighboring blocks, block's data size is enlarged and overlaps adjacent blocks; for each common quadrangle between two adjacent blocks of size $(n+3)^3$ each, $2n^2$ of voxels are duplicated. This allows parts of the mesh in each block to be generated independently of the others.

In the texture, we used a single GL_RED channel, where the data is stored in GL_FLOAT16 format. In order to allow reading distance field values, each point of the texture is sampled using linear filtering. This is done automatically, and the values are calculated on the basis of neighboring texels. Mipmaps for the textures are not generated.

5.2 Mesh Generation (Pass 1)

In the first pass calculations are performed using vertex shader and geometry shader. Then, using a transform feedback, generated indices are stored in the output buffer. Input and output data are presented in Table 1.

IN
Vertex buffer containing calculation points $p \in \{[0,n-1]\times[0,n-1]\}$ *type:* GL_ARRAY_BUFFER *size:* $(n-1)^2$ *sharing:* one per all blocks
Texture with volumetric data *type:* GL_TEXTURE_3D *size:* $(n+3)^3$ *sharing:* one per single block
OUT
Buffer with indices of generated quads *type:* GL_ELEMENT_ARRAY_BUFFER *sharing:* one per single block

Table 1: Input and output data of the first pass

The subsequent steps of the algorithm are as follows:

1. Data from the vertex buffer containing the calculation points is rendered $n-1$ times, using the instance rendering.

2. For a calculation point p three points q,r,s are created in the vertex shader.

3. For each of the points p,q,r,s a volumetric texture is sampled in order to retrieve implicit function value. Texture coordinates t are calculated according to formula 5.

$$t = \frac{p+2}{n+3} \tag{5}$$

4. Subsequently three edges pq, pr, ps are created and according to formula 3 it is checked whether the values of implicit function at the endpoints have different signs.

5. For each edge, if the sign changes, there is a flag set, which specifies whether the quadrilateral is generated or not, and what is its orientation (zero means that the quadrilateral is not generated). Generated flags along with the coordinates of the point p are forwarded to the geometry shader.

6. In the geometry shader, for each non-zero flag there a vertex containing four indices of generated quadrangles is established. Indices are calculated on the basis of the flag f and coordinates p. For example, quadrangle, which normal is consistent with the direction of the edge pq, is defined by (i_1,i_2,i_4,i_3), where

$$\begin{aligned} i_1 &= p_x n^2 + p_y n + p_{z+1} \\ i_2 &= p_x n^2 + p_{y+1} n + p_{z+1} \\ i_3 &= p_x n^2 + p_{y+1} n + p_z \\ i_4 &= p_x n^2 + p_y n + p_z \end{aligned}$$

7. Using the feedback transformation these indices are stored directly in the index buffer, used in the next pass. Number of saved indices is queried using an OpenGL query mechanism.

5.3 Rendering (pass 2)

The second pass is responsible for smoothing of the generated mesh and its rendering. All programmable shader units, i.e. vertex, fragment and geometry shaders are used. The input data is presented in Table 2.

Subsequent steps of the second pass of the algorithm are as follows:

1. The data from the vertex buffer is rendered using the indices as primitives of the GL_LINES_ADJACENCY type. This type was chosen because it is the only type that can render

IN
Vertex buffer which contains all potential vertices, such as: $u \in \{[0,n] \times [0,n] \times [0,n]\}$. *type:* GL_ARRAY_BUFFER *size:* n^3 *sharing:* one per all blocks
Buffer with indices for generated quadrangles *type:* GL_ELEMENT_ARRAY_BUFFER *sharing:* one per single block
Texture with volumetric data *type:* GL_TEXTURE_3D *size:* $(n+3)^3$ *sharing:* one per single block

Table 2: Input data for the second pass of the algorithm

primitives indexed by four indices (in OpenGL version 3.0 or higher it is not possible to render quadrangles).

2. Then in the vertex shader for each vertex u, a normal vector n is calculated, on the basis of the gradient map. Normal value is obtained by sampling the volumetric data texture in the neighboring six texels in x, y, z directions.

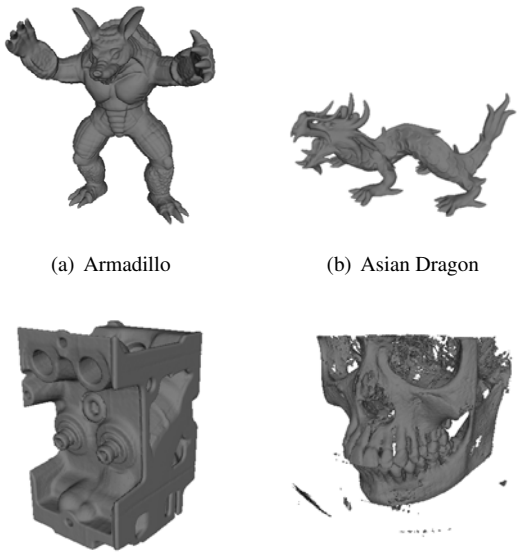
3. On the basis of the direction of the normal and the density function value, the point u is moved in the direction of the contour, according to formula 4. Due to the fact that the value of the density function is calculated as the average of neighboring texels at the point u, for small values of n it is required to perform a smoothing of the mesh in an iterative manner.

4. Vertices calculated in this way are sent to the geometry shader, in which there is a change of type done, from "lines adjacency" into "triangle strip".

5. The last step is to display a completed mesh, during which the associated shading calculations are performed in the fragment shader. In case when mesh rendering is not required, but the mesh is needed for further calculations, smoothed values of the vertices can be stored in the output buffer using a transform feedback.

6 RESULTS

All calculations were performed on an AMD Phenom II X6 1090T 3.2GHz computer with an nVidia GeForce GTX 460 graphics card. Figure 3 presents datasets used in tests; Armadillo and Asian Dragon from *The Stanford 3D Scanning Repository*, Engine and Skull from `http://www.volvis.org/`. In addition, three-dimensional animated Perlin noise has been used as a dynamic, time-dependent data. All tests were run for different n on one block of data.



(a) Armadillo (b) Asian Dragon

(c) Engine (d) Skull

Figure 3: Datasets used for tests

Table 3 lists times for mesh generation and rendering for different block sizes and different data sets. All meshes were regenerated from volumetric data each frame; moreover, calculations for time-dependent Perlin noise data were also performed before mesh generation every frame. Generated meshes for the Armadillo dataset for different block sizes are presented on figure 4, results for noise dataset are presented on figure 6.

dataset	n	triangle count	fps
Armadillo	64	14180	1100
Armadillo	128	91958	208
Armadillo	256	494880	28
Asian Dragon	64	5864	1180
Asian Dragon	128	41578	234
Asian Dragon	256	229840	30
Engine	256	592884	29
Skull	256	1699526	23.6
Perlin Noise	128	180k-246k	60

Table 3: Results for different block sizes and data sets

Table 4 presents results for our method compared to [Goe05, JC06]. The Engine and Skull datasets were used, no preprocessing were performed for these data. As it can be seen our algorithm performs much faster, however, if all methods would be run on the same hardware configuration, the difference could be less significant. Both methods of [Goe05] and [JC06] were tested on nVidia GeForce 6800GT graphics card.

The Marching Cubes algorithm described in [Gei07] seems to execute faster than [Goe05, JC06] methods but no measurable results were published. Authors of [Gei07] claims that their algorithm executes in interac-

(a) $n = 64$ (b) $n = 128$ (c) $n = 256$

Figure 4: Generated mesh for the Armadillo dataset

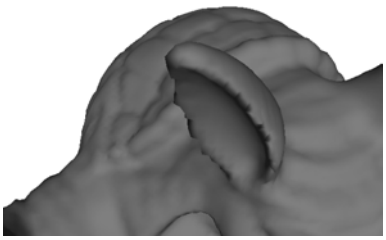
tive frames but mesh generation are not performed each frame.

dataset	size	method	fps
engine	256x256x110	[Goe05]	3.6
engine	256x256x128	[JC06]	2.8
engine	256x256x256	our method	29
skull	256x256x225	[Goe05]	2.4
skull	256x256x256	[JC06]	1.5
skull	256x256x256	our method	23.6

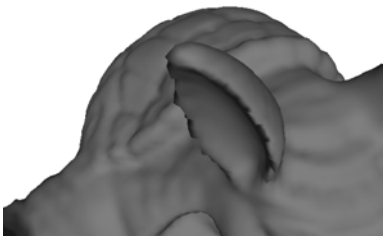
Table 4: Results compared to previous GPU-based methods

7 MESH IMPROVEMENTS

The presented method works well for smooth surfaces, such as the ones presented in figure 6. In case of surfaces with sharp edges we see artifacts as it is shown in figure 5(a).



(a) Artifacts on sharp edges



(b) Smoothed mesh

Figure 5: Mesh improvements due to smoothing

In order to improve visual quality of the generated surface, in the second pass of the algorithm a smoothing process based on normals is performed about 10 times. Next, during transforming quadrangles into strips of triangles, the quadrangles are divided along the shorter diagonal. The last step of smoothing the mesh is computation of normals for every pixel in fragment shader, on the basis of volumetric data. Normals are calculated in the same way as for the mesh vertices. The smoothed mesh is presented in Figure 5(b).

8 CONCLUSION AND FUTURE WORK

In this paper we present a real-time algorithm for generating a three-dimensional mesh fully based on volumetric data. This method has been optimized for graphics processors and provides a significant advantage over the already existing solutions for conventional processors. The presented algorithm is also very well suited for the visualization of dynamic data, because the calculations carried out do not need to know the state of the algorithm from previous time steps.

With the resulting performance, practical application of the algorithm in digital sculpting software, earth-moving machineries simulators and computer games is fully possible. The tests show a significant advantage of GPUs. The volumetric data representation that has been used allows also for efficient data modification using GPUs.

As part of further work on the algorithm it would be reasonable to add support for levels of detail (LOD), so as to enable the process to connect continuously adjacent blocks containing cubes of different sizes and densities.

The second issue is to optimize the algorithm by an additional parallelization and simultaneous calculation carried out for 4 blocks. It would be possible in the case of using all four available texture channels. As a result, it would be possible to generate meshes for the four blocks at the same time.

9 ACKNOWLEDGEMENTS

The work described in this paper was partially supported by The European Union by means of European

Social Fund, PO KL Priority IV: Higher Education and Research, Activity 4.1: Improvement and Development of Didactic Potential of the University and Increasing Number of Students of the Faculties Crucial for the National Economy Based on Knowledge, Subactivity 4.1.1: Improvement of the Didactic Potential of the AGH University of Science and Technology "Human Assets", No. UDA-POKL.04.01.01-00-367/08-00.

One of us (WA) kindly acknowledges partial support by Ministry of Science and Higher Education in Poland, project No. N N519 443039.

10 REFERENCES

[Blo88] J. Bloomenthal. Polygonization of implicit surfaces. *Comput. Aided Geom. Des.*, 5(4):341–355, 1988.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22, New York, NY, USA, 2009. ACM.

[GDL+02] Benjamin Gregorski, Mark Duchaineau, Peter Lindstrom, Valerio Pascucci, and Kenneth I. Joy. Interactive view-dependent rendering of large isosurfaces. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 475–484, Washington, DC, USA, 2002. IEEE Computer Society.

[Gei07] Ryan Geiss. *Generating Complex Procedural Terrains using the GPU*, chapter 1. GPU Gems. Addison Wesley, 2007.

[Gib98] Sarah F. Frisken Gibson. Constrained elastic surface nets: Generating smooth surfaces from binary segmented data. In *MICCAI '98: Proceedings of the First International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 888–898, London, UK, 1998. Springer-Verlag.

[Goe05] F. et al. Goetz. Real-time marching cubes on the vertex shader. In *In Proceedings of Eurographics 2005, Dublin, Ireland , pp. 5-8*, 2005.

[HSIW96] A. Hilton, A. J. Stoddart, J. Illingworth, and T. Windeatt. Marching triangles: range image fusion for complex object modelling. In *Proc. Conf. Int Image Processing*, volume 1, pages 381–384, 1996.

[JC06] Gunnar Johansson and Hamish Carr. Accelerating marching cubes with graphics hardware. In *In CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research, ACM*, page 378. Press, 2006.

[JLSW02] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 339–346, New York, NY, USA, 2002. ACM.

[KKDH07] Michael Kazhdan, Allison Klein, Ketan Dalal, and Hugues Hoppe. Unconstrained isosurface extraction on arbitrary octrees. In *SGP '07: Proceedings of the fifth Eurographics symposium on Geometry processing*, pages 125–133, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[KOKK06] Takashi Kanai, Yutaka Ohtake, Hiroaki Kawata, and Kiwamu Kase. Gpu-based rendering of sparse low-degree implicit surfaces. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, GRAPHITE '06, pages 165–171, New York, NY, USA, 2006. ACM.

[KOR08] John Kloetzli, Marc Olano, and Penny Rheingans. Interactive volume isosurface rendering using bt volumes. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 45–52, New York, NY, USA, 2008. ACM.

[KW05] P. Kipfer and R. Westermann. Gpu construction and transparent rendering of isosurfaces. In *Vision, Modeling, and Visualization (VMV 2005), Erlangen, Germany, Nov. 16-18*, 2005.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21:163–169, August 1987.

[LK10] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 55–63, New York, NY, USA, 2010. ACM.

[Nie04] Gregory M. Nielson. Dual marching cubes. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 489–496, Washington, DC, USA, 2004. IEEE Computer Society.

[OBA05] Yutaka Ohtake, Alexander Belyaev, and

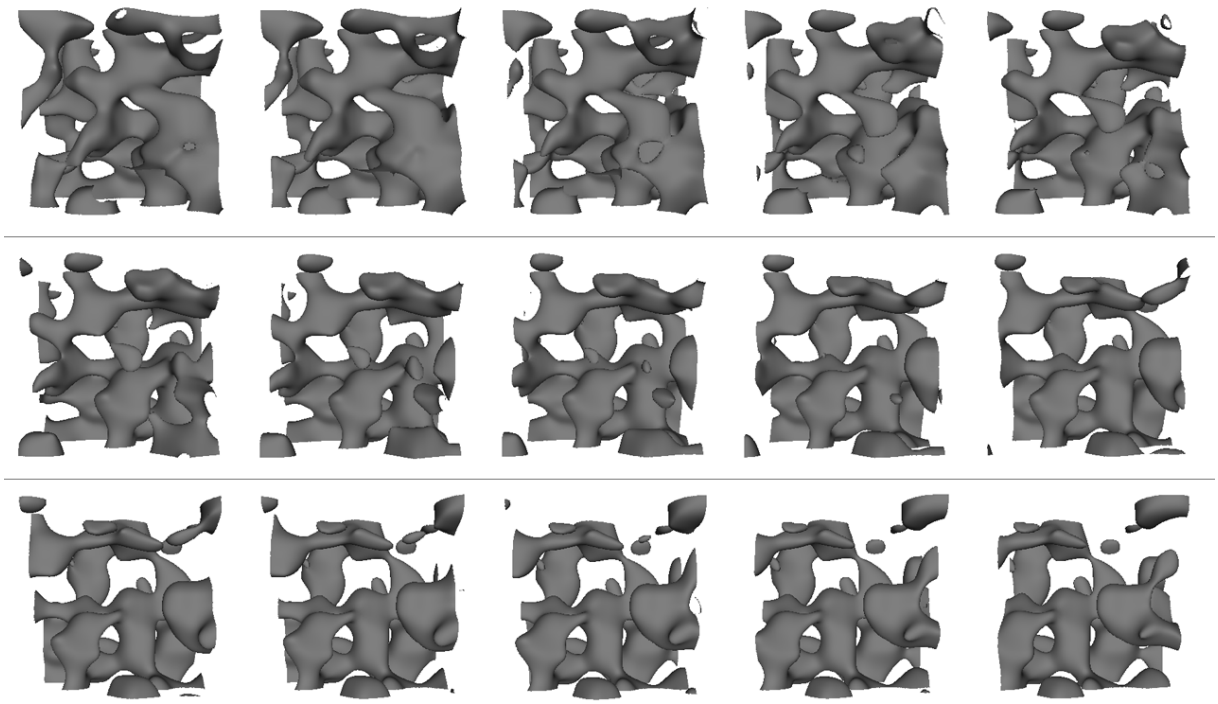


Figure 6: Generated meshes for Perlin noise dataset for 15 frames

Marc Alexa. Sparse low-degree implicit surfaces with applications to high quality rendering, feature extraction, and smoothing. In *Proceedings of the third Eurographics symposium on Geometry processing*, SGP '05, Aire-la-Ville, Switzerland, Switzerland, 2005. Eurographics Association.

[OR97] M. Ohlberger and M. Rumpf. Hierarchical and adaptive visualization on nested grids. *Computing*, 59(4):365–385, 1997.

[TPG99] G. M. Treece, R. W. Prager, and A. H. Gee. Regularised marching tetrahedra: improved isosurface extraction. *Computers and Graphics*, 23:583–598, 1999.

[TSD07] Natalya Tatarchuk, Jeremy Shopf, and Christopher DeCoro. Real-time isosurface extraction using the gpu programmable geometry pipeline. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 122–137, New York, NY, USA, 2007. ACM.

[WKE99] R. Westermann, L. Kobbelt, and T. Ertl. Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces. *The Visual Computer*, 15:100–111, 1999.

A Layered Depth-of-Field Method for Solving Partial Occlusion

David C. Schedl*
david.schedl@cg.tuwien.ac.at

Michael Wimmer*
wimmer@cg.tuwien.ac.at

*Vienna University of Technology, Austria

ABSTRACT

Depth of field (DoF) represents a distance range around a focal plane, where objects on an image are crisp. DoF is one of the effects which significantly contributes to the photorealism of images and therefore is often simulated in rendered images. Various methods for simulating DoF have been proposed so far, but little tackle the issue of partial occlusion: Blurry objects near the camera are semi-transparent and result in partially visible background objects. This effect is strongly apparent in miniature and macro photography. In this work a DoF method is presented which simulates partial occlusion. The contribution of this work is a layered method where the scene is rendered into layers. Blurring is done efficiently with recursive Gaussian filters. Due to the usage of Gaussian filters big artifact-free blurring radii can be simulated at reasonable costs.

Keywords:
depth of field, rendering, real-time, layers, post-processing

1 INTRODUCTION

DoF represents a distance range around a focal plane in optic systems, such as camera lenses. Objects out of this range appear to be blurred compared to sharp objects in focus. This effect emphasizes objects in focus and therefore is an important artistic tool in pictures and videos.

People in the field of computer graphics aim for the ambitious goal of generating photo-realistic renderings. Depth of Field is one effect which significantly contributes to the photorealism of images because it is an effect that occurs in most optical systems. In computer renderings, the pinhole-camera model, which relies upon the assumption that all light-rays travel through one point before hitting the image plane, is used. Therefore, there is no focus range and no smearing occurs, resulting in a crisp image. However, in real-life optical systems—such as the eye or photographic cameras—sharp images are only produced if the viewed object is within a certain depth range: the depth of field.

DoF can be simulated very accurately by ray tracing, but the rendering of accurate DoF effects is far from interactive frame rates. For interactive applications, the effect has to be approximated in real time. Therefore, most approaches use fast post-processing techniques and sacrifice visual quality, causing artifacts. Common techniques to produce the DoF effect use an approach where pixels get smeared according to their circle of confusion (CoC) [25]. The CoC depends on the distance of objects and the lens parameters. One artifact in post-processing approaches is *partial occlusion*: An object in-focus occluded by an out-of-focus object should be partly visible at the blurred object borders of the front object. In computer graphics, the used pinhole camera model in combination with depth testing leads to a dismissing of background pixels. Real optical systems use a finite aperture camera model where light rays from occluded objects can hit the image sensor. Figure 1 shows this effect next to a pinhole rendering.

(a) (b)

Figure 1: A pinhole rendering of the scene *Dragons* resulting in a crisp image (a). Simulating shallow depth-of-field with the proposed method partly reveals occluded scene content (b). Note how the tongue of the dragon almost vanishes.

In this paper, we present an approach to tackling the partial occlusion problem. By rendering the scene with depth peeling [11], occluded pixels can be retrieved (Section 3.1). This occluded scene information is used

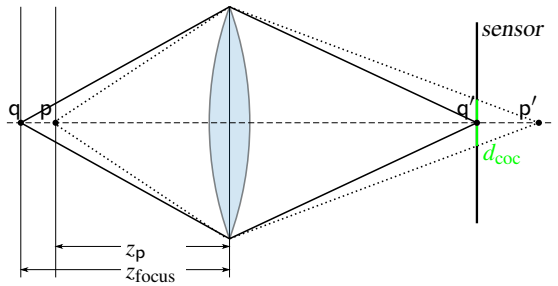


Figure 2: The model of a thin lens and how the points q (in-focus) and p (out-of-focus) are projected onto the image sensor (inspired by [25]).

to overcome the problem of partial occlusion. Rendered fragments are weighted, based on their depth, into layers (section 3.2), where each layer is blurred uniformly (section 3.3). Previous such layered DoF methods produced artifacts due to the layer splitting. We avoid most of these artifacts by smoothly decomposing layers and additional scene information. After blurring, the layers are composed by blending, thus producing renderings with convincing partial occlusion effects.

2 PREVIOUS WORK

DoF is an effect caused by the fact that optical lenses in camera systems refract light rays onto the image sensor, but fail to produce crisp projections for all rays. Figure 2 shows a schematics of a thin lens model and how rays are refracted. Although modern optical systems use a set of lenses, for the purpose of explaining DoF, a single lens is sufficient. Hypothetically, a sharp image point will only appear on the image plane from an object exactly in focus, located at z_{focus} (see figure 2). In practice, because of limitations of the human eye, objects within an acceptable sharpness are recognized as sharp. Objects out of the DoF range are projected as circles on the image plane. The diameter of this so-called circle of confusion (CoC) can be calculated as

$$d_{\text{coc}}(z,f,N,z_{\text{focus}}) = \left| \frac{f^2\,(z - z_{\text{focus}})}{zN\,(z_{\text{focus}} - f)} \right|, \qquad (1)$$

where z is the distance to the object in front of the lens, f is the focal length of the lens, N is the f-stop number, and z_{focus} is the focus distance [25].

While the blurring is produced as an imperfection in optical systems, computer renderings usually produce a crisp image. Therefore DoF has to be simulated by specific rendering methods [1, 2, 10, 3].

Methods operating in object space simulate rays that do not go through the center of the lens. These methods include distributed ray tracing [8] and the accumulation buffer method [12], which both produce high-quality results but fail to deliver real-time frame rates.

Faster methods are based on the idea of rendering the scene with a pinhole camera model and simulating the DoF effect via post processing, leading to few or no changes in the rendering pipeline. The first method discussed by Potmesil and Chakravarty in 1981 presented equation 1, the formula for calculating the CoC [25]. Most modern methods (including this one) are based on this work.

Methods using graphic cards for acceleration use pyramid methods, Poisson sampling or a combination of both [24, 26, 27, 13, 22]. Isotropic filters lead to intensity leaking artifacts, where colors from in-focus foreground pixel bleed on the out-of-focus background. Cross-bilateral filters or heat diffusion ([6, 14]) can be used to overcome this artifact, but this introduces other issues like discontinuity artifacts: Out-of-focus objects have sharp boundaries although the object itself is blurred.

The partial occlusion artifact is apparent in all previously mentioned methods. Rasterization techniques do not store occluded fragments, therefore it is not possible to accurately simulate transparency caused by out-of-focus smearing. To fully simulate this effect, occluded information has to be either stored or interpolated in layers. Layered DoF methods compose scene fragments into layers depending on fragment depth. With this representation it is possible to store or interpolate occluded scene content. Furthermore it is possible to uniformly blur each layer. One prominent artifact in layered DoF methods are discretization artifacts: Layers get blurred and therefore object borders are smeared out. When this smeared-out layer is blended with the other layers, the smeared border region appears as a ringing artifact at object borders due to the reduced opacity. In [4, 5], the authors investigate such artifacts. One way to avoid these ringing artifacts is presented in [18], where occluded scene information is interpolated before layers are created. Blurring and interpolation is done by a pyramidal method, which approximates a Gaussian filter. The author presents a variation of this method in [17], where the costly interpolation steps are left out and different filters are used. However, these methods fail to correctly solve the partial occlusion problem, because hidden scene information is only interpolated and does not represent any actual scene content.

The DoF methods [21, 15] are able to solve partial occlusion by generating CoC-sized splats for each pixel. However, these methods come with additional costs for sorting fragments, making them impractical for complex scenes.

In [19], layered rendering is used to generate a layered representation. The DoF effect is then generated by ray-traversing these layers. Rays are scattered across the aperture of a virtual lens, thus avoiding the previously mentioned discretization artifacts. An improvement is discussed in [20], where the layers are generated by

depth peeling and ray-tracing is done differently. Furthermore various lens effects (e.g., chromatic aberration and lens distortion) can be simulated. However, the method needs preprocessing previously to ray intersecting and needs back faces to be rendered. If the number of rays is not sufficient both methods produces noise and aliasing artifacts. Especially for strong blurs many rays have to be used, resulting in non-interactive rates.

For a solid approximation of partial occlusion, a layered scene representation, storing occluded fragments, has to be used. The approach presented in the following section is a layered DoF method which produces convincing partial occlusion effects while vastly avoiding the discussed issues.

3 METHOD

The method proposed in this paper decomposes the scene into *depth layers*, where each layer contains pixels of a certain depth range. The resulting layers are then blurred with a filter that is sized according to the distance from the focal point, and then composited. This approach handles partial occlusion, because hidden objects are represented in more distant layers and contribute to the compositing.

One way to generate the K layers would be to render the scene K times, with near- and far planes adjusted to cover the desired depth range of the layer. However, this leads to two problems: first, rendering the scene K times is too expensive for interactive applications, and second, discretization artifacts would appear due to the hard layer borders. In this paper, we solve both problems:

In order to avoid rendering the scene K times, we use depth peeling to generate a number M of *occlusion layers* (also named *buffers* in the following), where $M < K$. Note that each occlusion layer can contain fragments from the full depth range of the scene, while a depth layer is bound by its associated depth range. We then generate the depth layers by decomposing the occlusion layers into the depth ranges, which is much faster than rendering each depth layer separately.

To avoid discretization artifacts, we do not use hard boundaries for each depth layer, but a smooth transition between the layers, given by *matting functions*.

Furthermore, we also propose a method for efficiently computing both the blur and the layer composition in one step.

Our method consists of the following steps:

1. Render the scene into M buffers, where I_0 and Z_0 contain the color and depth from an initial pinhole rendering. The buffers $I_1 \dots I_{M-1}$ and $Z_1 \dots Z_{M-1}$ store peeled fragments from front to back.

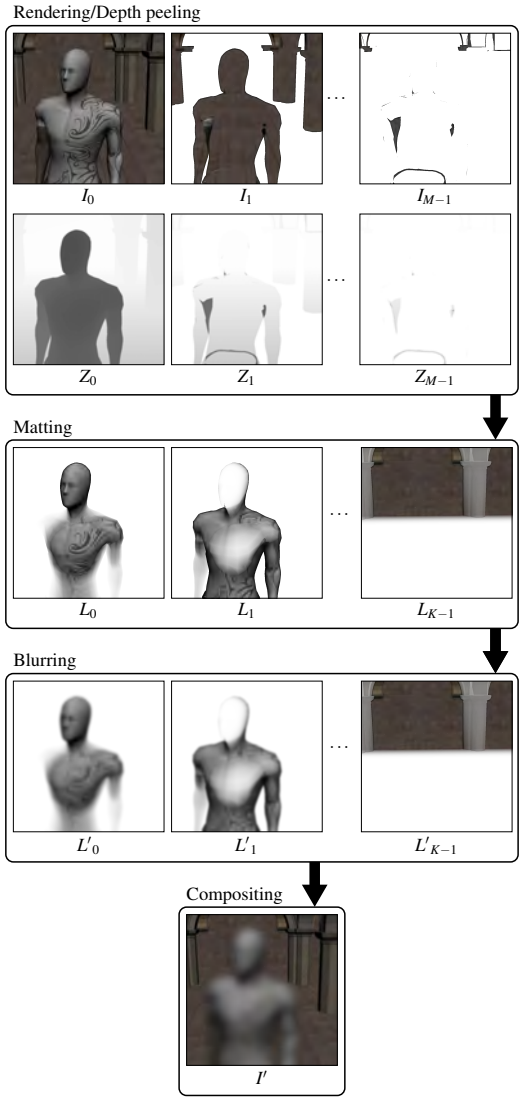


Figure 3: A overview of the proposed method in this work: The scene is rendered into color buffers I_0 to I_{M-1} and depth buffers Z_0 to Z_{M-1} by depth peeling . The color buffers are decomposed into K layers L_0 to L_{K-1} by a depth-dependent matting function. The decomposed layers get blurred by their CoC and composed. Note that the final algorithm combines the blurring and composing step, which is simplified in this figure.

2. Decompose the fragments of the input buffers into K depth layers L_0 to L_{K-1}, based on a matting function and the fragments' depth.

3. Blend and blur each layer L_k onto the buffers I'_{front}, I'_{focus} or I'_{back}, which are composed back-to-front afterwards.

Figure 3 outlines the above described algorithm.

3.1 Rendering

Rendering is done by depth peeling [11]. For depth peeling, first a 3D scene is rendered into a buffer storing the color I_0 and the depth Z_0 of a rendering, shown in figure 3. Then the scene is rendered a second time

into new buffers I_m and Z_m while projecting the previous depth buffer Z_{m-1} onto the scene. A fragment p gets rejected if its depth z_p has the same or smaller depth than the previously rendered fragment, stored in I_{m-1} and Z_{m-1}. This means that only previously occluded fragments are stored in I_m and Z_m. This is done iteratively until M layers are retrieved. If a fragment is rejected, it is "peeled away," revealing objects behind the first layer. Although there are faster peeling methods (e.g., [23]), we rely on [11], because peeling can be done iteratively from front to back.

3.2 Scene decomposition

The input images $I_0 \dots I_{M-1}$ are decomposed into K layers $L_0 \dots L_{(K-1)}$ by matting functions $\omega(z)$ and $\dot{\omega}$:

$$L_k = \Big(I_0 \cdot \omega_k(Z_0)\Big) \oplus \Big(I_1 \cdot \dot{\omega}_k(Z_1)\Big) \dots \oplus \Big(I_{M-1} \cdot \dot{\omega}_k(Z_{M-1})\Big). \quad (2)$$

The functions $\omega_k(z)$ and $\dot{\omega}_k(z)$ denote the matting function for the layer L_k and $A \oplus B$ denotes alpha-blending A over B.

3.2.1 Matting functions

The matting function ω_k was introduced in [18] and guarantees a smooth transition of objects between layers, while $\dot{\omega}_k$ retains a hard cut at the back layer boundaries to avoid situations where background fragments would be blended over foreground layers. The formulas are

$$\dot{\omega}_k(z) = \begin{cases} \frac{z - z_{k-2}}{z_{k-1} - z_{k-2}} & \text{for} \quad z_{k-2} < z < z_{k-1}, \\ 1 & \text{for} \quad z_{k-1} \le z \le z_k, \\ 0 & \text{otherwise}, \end{cases} \quad (3)$$

and

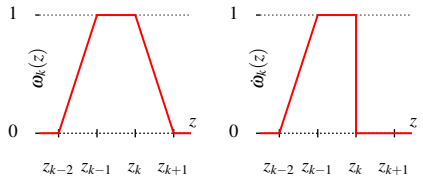
$$\omega_k(z) = \begin{cases} \frac{z_k - z}{z_k - z_{k+1}} & \text{for} \quad z_k < z < z_{k+1}, \\ \dot{\omega}_k(z) & \text{otherwise}, \end{cases} \quad (4)$$

where z_{k-2} to z_{k+1} defines anchor points for the layer boundaries. A plot of the functions is shown in figure 4. Special care has to be taken when matting the front L_0 and back L_{K-1} layer, where the boundaries are set to $z_{-2} = z_{-1} = z_0 = -\infty$ and $z_{K-1} = z_K = \infty$, respectively.

3.2.2 Layer boundaries

The layer matting relies on anchor points. Similarly to [18], the boundaries are spaced according to the filter size of the blurring method (further explained in section 3.3). Potmesil's formula for calculating the CoC (equation 1) can be rearranged to calculate a depth z based on a given CoC d. Since d_{coc} is non-injective, there are two possible results of this inversion:

$$d_{\text{coc}}^{-1}(d) = \big(D_1(d), D_2(d)\big) \quad (5)$$



(a) (b)

Figure 4: The matting functions ω_k (a) and $\dot{\omega}_k$ (b) with exemplary depth coordinates z_{k-2} to z_{k+1}.

with

$$D_1(d) = \frac{z_{\text{focus}} \cdot f^2}{f^2 + d \cdot N \cdot (z_{\text{focus}} - f)}, \quad (6)$$

$$D_2(d) = \frac{z_{\text{focus}} \cdot f^2}{f^2 - d \cdot N \cdot (z_{\text{focus}} - f)}. \quad (7)$$

With equation 5, the depth of anchor points can be calculated by using d_{coc}as input parameter, calculated by the filter size of the blurring method. Note that $D_2(d)$, $d \in \mathbb{R}^+$ is only applicable as long as

$$d < \frac{f^2}{N \cdot (z_{\text{focus}} - f)}. \quad (8)$$

The anchor point furthest away from the camera, z_{K-1}, is limited by this constraint. An anchor pointz_k is placed at the average CoC of the layers L_k and L_{k+1}. Thus

$$z_k = \begin{cases} D_1\left(\frac{d_k + d_{k+1}}{2}\right) & \text{for} \quad k < k_{\text{focus}}, \\ D_2\left(\frac{d_k + d_{k+1}}{2}\right) & \text{for} \quad k \ge k_{\text{focus}}, \end{cases} \quad (9)$$

where k_{focus} is the index of the layer in focus and d_k and d_{k+1} are the CoCs of the layers L_k and L_{k+1} respectively. The layer's CoC d_k is given by the blur radius for a discrete layer L_k, determined by the blurring method (see section 3.3).

3.2.3 Determining the number of layers

The depth of rendered fragments in the scene should lie within the depth range of the closest and furthest anchor points (z_{K-1} and z_0). Therefore enough anchor points to cover the scene have to be generated. This can be done manually or automatically. One naive automatic approach would be to use the near and far clipping planes, resulting in the highest possible depth range, which usually is not present in a scene. A better approach is to use hierarchical N-Buffers for determining the minimum and maximum depth values within the view frustum [9].

3.3 Blurring and Composition

We use Gaussian filters for blurring, because they can be separated, recursively applied and produce smooth results. The mapping from CoC to the standard deviation σ of a Gaussian kernel is chosen empirically as

$$d_{\text{pix}} = 4\sigma. \quad (10)$$

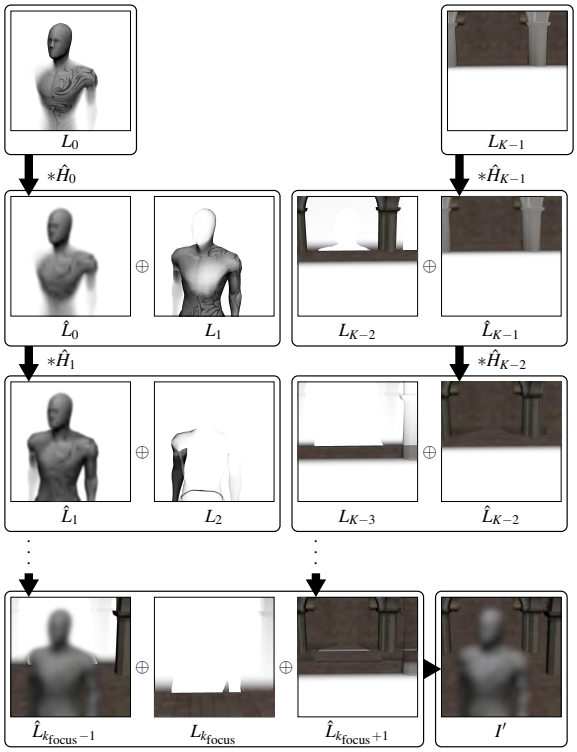


Figure 5: Overview of the combined blurring and composition steps: The layers L_0 to L_{K-1} are blended iteratively. Between each blending step the composition is blurred. Layers in front of the focus layer $L_{k_{\text{focus}}}$ and layers behind the focus layer are composed separately. Those are combined in a final step into the result I'.

Note that d_{pix} is the d_{coc} in screen coordinates and has to be transformed into the world coordinate system. Each layer is blurred by a Gaussian kernel H_k with the standard deviation σ_k as

$$L'_k = L_k * H_k, \tag{11}$$

where $*$ denotes a convolution.

Instead of convolving each layer separately (shown in figure 3 for illustrative reasons), a cascaded approach is chosen. Between each blur, one layer is blended onto the composition.

Layers in front and behind the in-focus layer have to be composed and blurred separately. Otherwise it is not possible to keep the correct depth ordering of the layers. The composition of the front layer starts by taking the layer closest to the camera (i.e., L_0) and blurring it with the filter kernel $\hat{H}_0$. In the next step this blurred layer is blended over the next closest layer (i.e., L_1) and afterwards blurred with $\hat{H}_1$. A schematic of the composition steps is shown in figure 5. Since a blurred layer L_k is blended over L_{k+1} and then blurred again, the effect of this method is that L_k is blurred by $\hat{H}_k$ and by $\hat{H}_{k+1}$. The iteration continues until the layer in-focus $L_{k_{\text{focus}}}$ is reached. In general, such recursive Gaussian filters produce the same result as blurring with one big Gaussian. The resulting filter sizes can be calculated by the Euclidean distance [7, chapter 8]. However, in our application the results differ due to occlusions within the layers.

Back layers are blurred similarly, starting with L_{K-1}. To keep the correct layer ordering, the layer closer to the camera (i.e., L_{K-2}) has to be blended over the previously blurred layer. The iteration is again continued until the layer in-focus is reached.

The number of blurring iterations for a layer L_k is given by $|k - k_{\text{focus}}|$. Calculating the final composition I' is done by

$$I' = \hat{L}_{k_{\text{focus}}-1} \oplus (L_{k_{\text{focus}}} \oplus \hat{L}_{k_{\text{focus}}+1}), \tag{12}$$

where

$$\hat{L}_k = \begin{cases} L_k & \text{for} \quad k = k_{\text{focus}} \\ L_k * \hat{H}_k & \text{for} \quad k = 0 \text{ and } k = K-1 \\ (\hat{L}_{k-1} \oplus L_k) * \hat{H}_k & \text{for} \quad k < k_{\text{focus}} \\ (L_k \oplus \hat{L}_{k+1}) * \hat{H}_k & \text{for} \quad k > k_{\text{focus}} \end{cases} \tag{13}$$

Results in section 4 are produced with a Gaussian filter kernel $\hat{H}_k$ with a standard deviation of $\hat{\sigma}_k$:

$$\hat{\sigma}_k = |k - k_{\text{focus}}|. \tag{14}$$

Various methods for calculating the filter size can be used. For Gaussians, the adequate (non-recursive) σ_k can be calculated by

$$\sigma_k = \begin{cases} 0 & \text{for} \quad k = k_{\text{focus}}, \\ \sqrt{\hat{\sigma}_k^2 + \sigma_{k+1}^2} & \text{for} \quad k < k_{\text{focus}}, \\ \sqrt{\hat{\sigma}_k^2 + \sigma_{k-1}^2} & \text{for} \quad k > k_{\text{focus}}, \end{cases} \tag{15}$$

where k is in the interval $[0, K-1]$.

3.3.1 *Normalization*

Due to the usage of matting functions ω and $\dot{\omega}$, resulting in expanded depth layers, and the usage of depth peeling, discretization artifacts as discussed in [5, 4] are mostly avoided. However, in some circumstances (e.g., almost perpendicular planes) such artifacts may still appear. We use premultiplied color values while matting and filtering. Therefore the composition can be normalized (divided by alpha), thus further minimizing discretization artifacts.

4 RESULTS

The proposed method was implemented in OpenGL and the shading language GLSL. Performance benchmarks are done on an Intel Core i7 920 CPU with a Geforce GTX 480 graphics card. Depth peeling uses a 32bit

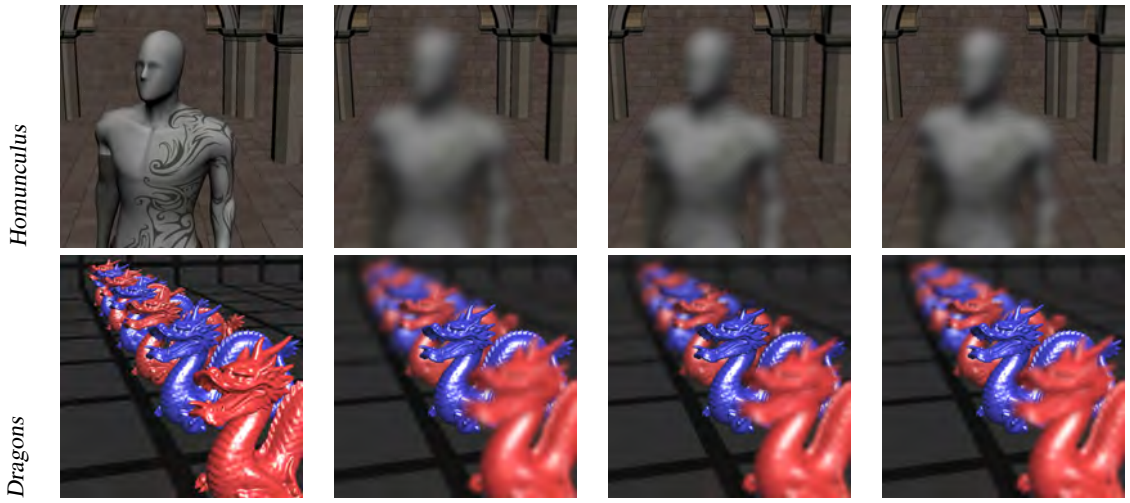


Figure 6: DoF effects produced with our method, the ray-traversal method ([20]) and the accumulation-buffer method. The first row shows the scene *Homunculus* (74k faces), and the second shows *Dragons* (610k faces). Renderings have the resolution 1024×1024. Note that there are sampling artifacts on the chest of the *Homunculus* scene in the accumulation and ray-traversal method, although there are 256 rays/views used. Our method avoids such artifacts by cascaded Gaussian filtering. Also note the partial-occlusion artifacts (e.g., in the second red dragon) in the ray-traversal method. The lens settings are $f = 0.1$, $N = 1.4$ and is focused at the stone wall in the back ($z_{\text{focus}} = 18.5$) for the *Homunculus* and at the foremost blue dragon ($z_{\text{focus}} = 3$) for the *Dragon* scene.

z-buffer to avoid any bias values due to z-buffer imprecisions.

We compare our method to the accumulation buffer-technique [12] and to a ray-traversal technique[20], because the first simulates high-quality DoF effects if enough views are sampled, while the latter method is a state-of-the-art method which handles partial occlusion correctly. The accumulation buffer technique is implemented in OpenGL, but does not use the accumulation buffer, because of precision issues when accumulating a high number of views. Instead, a 32bit-per-channel float texture is used for accumulation. The ray-traversal method was implemented in OpenGL and GLSL. Although there are some tweakable parameters, the authors give little information about their configuration. So for the intersection tests of rays, we use 100 steps for the linear search and 5 for the binary search in normalized device coordinates. A hierarchical N-Buffer is used to decrease the depth range for each rays. In our implementation, we decrease the depth range for each ray individually—while the original authors packed rays—and we use 4 regular depth-peeling layers, without any optimizations. Additionally, the ray-traversal method requires closed scene objects and back-face culling to be turned off, for reliable intersection testing. This introduces additional rendering costs and decreases the usable peeling layers to only 2. The authors propose ways to overcome the latter limitation. However, in our implementation we use the simplified version containing only two peeling layers and their backface counterparts. For both reference methods we use a Gaussian distribution for lens samples positioning.

The methods are applied to the scenes *Homunculus* (74k triangles) and *Dragons* (610k triangles), shown in figure 6. Performance comparisons are shown in table 1. Rendering costs for the accumulation-buffer method are basically the costs for one scene rendering multiplied by the number of views. Our method is, apart from depth peeling, independent of the scene complexity, and faster than the ray-traversal method, even when that method uses only 32 rays, resulting in sampling artifacts. Our method is faster at processing the *Dragons* scene, although the scene *Homunculus* has fewer triangles. This can be explained by the distribution of the depth layers and the resulting amount of blur. In the *Homunculus* scene there are more highly blurred foreground layers, resulting in overall more rendering costs than in *Dragons*, where the layers are more, but evenly spread. Note that although scene *Dragons* has more triangles than *Homunculus*, it is rendered faster due to shading without textures and the use of vertex-buffer objects.

We currently store all sub-images on the graphics card—for convenience and debug reasons—resulting in heavy memory usage. However, additional depth layers ($\hat{L}_0$ to $\hat{L}_{K-1}$) can be avoided by applying the process-queue (matting, blurring, composing) in one buffer, which would decrease memory consumption.

	our (DP/matting/blur) Total		(DP/ray-traversal) Total			accum.
	cascaded	non-cascaded	256	128	32 rays	256 views
Homunculus (74k tri.)	(46/5/51)102	(46/5/95)146	(58/1290)1348	(48/643)691	(48/140)188	4809
Dragons (610k tri.)	(40/7/51)98	(40/8/85)133	(69/1374)1443	(69/685)754	(59/152)211	4163

Table 1: Performance comparisons, in ms, of our method (cascaded and non-cascaded blurring) with the ray-traversal method ([20]) and the accumulation-buffer method for the scenes *Homunculus* and *Dragons*. Renderings have the resolution 1024×1024 and 4 Depth-peeling iterations (DP) have been used.

5 CONCLUSION AND FUTURE WORK

We have presented a depth-of-field post-processing method with the aim of overcoming the partial occlusion artifact. The contribution of this work is a simple, efficient and GPU-friendly method. We combine depth-peeling with improved matting functions to avoid the overhead of rendering to a high number of depth layers. Furthermore we use high-quality Gaussian filters in a recursive way, which has not been done—to our knowledge—in DoF methods before. With the usage of Gaussian filters, high blur radii can be simulated, where even the reference methods start to produce sampling artifacts. We have shown that those DoF effects can be produced at frame rates that are significantly higher than previous methods, making high-quality DoF available for interactive applications.

One important step for the correct handling of partial occlusion is depth peeling, which is frequently used to resolve transparency issues, thus making the method hardly usable for interactive applications like games.

Currently we use Gaussian filters, which are separable and can be computed efficiently while delivering artifacts-free images. The usage of cascaded filters while composing the DoF effect slightly alters the produced image, but results in better performance. If higher frame rates are needed and visual quality can be sacrificed faster blurring methods (e.g., box, pyramid filters) can be used.

The composition by alpha-blending is simple and efficient, thus leading to faster results when compared to current methods like [20]. Layering discretization artifacts known from other methods are mostly avoided by matting, depth peeling and normalization.

Wide-spread anti-aliasing methods (i.e., MSAA) cannot be easily enabled for our method. However, image-based anti-aliasing methods (such as MLAA or FXAA)—which are becoming more popular due to the wide usage of deferred shading—can be applied.

Currently, layers are split based on the d_{coc} of fragments and on the chosen blurring method. This might result in empty layers. Decomposition could be optimized by using clustering methods, such as k-means clustering, as proposed in [21, 16]. With the use of clustering, layer borders could be tailored to the pixel density in scenes and empty layers could be avoided. However, clustering is a costly process and therefore only applicable for off-line rendering.

One further field of investigation would be the impact of correct partial occlusion rendering on human perception. We think that a correct handling of partial occlusion in combination with gaze-dependent focusing (e.g., with an eye-tracker) would result in deeper immersion of the user.

6 ACKNOWLEDGMENTS

Thanks to Juergen Koller for providing the Homunculus model. The used Dragon and Sponza models are courtesy of Stanford Computer Graphics Laboratory and Marko Dabrovic.

7 REFERENCES

[1] Brian A. Barsky, Daniel R. Horn, Stanley A. Klein, Jeffrey A. Pang, and Meng Yu. Camera models and optical systems used in computer graphics: Part I, Object based techniques. Technical report, University of Berkeley, California, USA, 2003.

[2] Brian A. Barsky, Daniel R. Horn, Stanley A. Klein, Jeffrey A. Pang, and Meng Yu. Camera models and optical systems used in computer graphics: Part II, Image-based techniques. Technical report, University of Berkeley, California, USA, 2003.

[3] Brian A. Barsky and Todd J. Kosloff. Algorithms for rendering depth of field effects in computer graphics. In *Proceedings of the 12th WSEAS international conference on Computers*, pages 999–1010, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[4] Brian A. Barsky, Daniel R. Tobias, Michael J.and Horn, and Derrick P. Chu. Investigating occlusion and discretization problems in image space blurring techniques. In *First International Conference on Vision, Video and Graphics*, pages 97–102, University of Bath, UK, July 2003.

[5] Brian A. Barsky, Michael J. Tobias, Derrick P. Chu, and Daniel R. Horn. Elimination of artifacts due to occlusion and discretization problems in image space blurring techniques. *Graphics Models*, 67(6):584–599, November 2005.

[6] Marcelo Bertalmio, Pere Fort, and Daniel Sanchez-Crespo. Real-time accurate depth of field using anisotropic diffusion and programmable graphics cards. In *Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, 3DPVT '04, pages 767–773, Washington, DC, USA, 2004.

[7] Wilhelm Burger and Mark J. Burge. Principles of Digital Image Processing: Advanced Techniques. To appear, 2011.

[8] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 137–145, New York, NY, USA, 1984. ACM.

[9] Xavier Décoret. N-buffers for efficient depth map query. *Computer Graphics Forum*, 24(3), 2005.

[10] J. Demers. Depth of field: A survey of techniques. In Fernand Randima, editor, *GPU Gems*, chapter 23, pages 375–390. Pearson Education, 2004.

[11] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA, 2001.

[12] Paul Haeberli and Kurt Akeley. The accumulation buffer: hardware support for high-quality rendering. *SIGGRAPH Computer Graphics*, 24:309–318, September 1990.

[13] Earl J. Hammon. Practical post-process depth of field. In Hubert Nguyen, editor, *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 28, pages 583–606. Addison-Wesley, 2007.

[14] Michael Kass, Lefohn Aaron, and John Owens. Interactive depth of field using simulated diffusion on a GPU. Technical report, Pixar Animation Studios, 2006.

[15] Todd J. Kosloff, Michael W. Tao, and Brian A. Barsky. Depth of field postprocessing for layered scenes using constant-time rectangle spreading. In *Proceedings of Graphics Interface 2009*, pages 39–46, Toronto, Canada, 2009.

[16] Todd Jerome Kosloff. *Fast Image Filters for Depth of Field Post-Processing*. PhD thesis, EECS Department, University of California, Berkeley, May 2010.

[17] Martin Kraus. Using Opaque Image Blur for Real-Time Depth-of-Field Rendering. In *Proceedings of the International Conference on Computer Graphics Theory and Applications : GRAPP 2011*, pages 153–159, Portugal, 2011. Institute for Systems and Technologies of Information, Control and Communication.

[18] Martin Kraus and Magnus Strengert. Depth-of-field rendering by pyramidal image processing. *Computer Graphics Forum*, 26(3):645–654, 2007.

[19] Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. Depth-of-field rendering with multiview synthesis. *ACM Transactions on Graphics (TOG)*, 28(5):1–6, 2009.

[20] Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. Real-time lens blur effects and focus control. *ACM Transactions on Graphics (TOG)*, 29(4):65:1–65:7, July 2010.

[21] Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. Real-time depth-of-field rendering using splatting on per-pixel layers. *Computer Graphics Forum (Proc. Pacific Graphics'08)*, 27(7):1955–1962, 2008.

[22] Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. Real-time depth-of-field rendering using anisotropically filtered mipmap interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):453–464, 2009.

[23] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. Single pass depth peeling via cuda rasterizer. In *SIGGRAPH 2009: Talks*, SIGGRAPH '09, New York, NY, USA, 2009. ACM.

[24] Jurriaan D. Mulder and Robert van Liere. Fast perception-based depth of field rendering. In *Proceedings of the ACM symposium on Virtual Reality Software and Technology*, VRST '00, pages 129–133, Seoul, Korea, October 2000. ACM.

[25] Michael Potmesil and Indranil Chakravarty. A lens and aperture camera model for synthetic image generation. In *Proceedings of the 8th Aannual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '81, pages 297–305, Dallas, Texas, USA, 1981. ACM.

[26] Guennadi Riguer. Real-time depth of field simulation. In Wolfgang F. Engel, editor, *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9*, pages 529–556. Wordware Publishing, October 2003.

[27] Thorsten Scheuermann and Natalya Tatarchuk. Improved depth of field rendering. In Wolfgang Engel, editor, *ShaderX^3: Advanced Rendering Techniques in DirectX and OpenGL*, pages 363–378. Charles River Media, 2004.

Journal of WSCG

Index