SIGRAD2004

The Annual SIGRAD Conference
Special Theme – Environmental Visualization
November 24th and 25th, 2004
University of Gävle, Gävle, Sweden,

Conference Proceedings

organized by

Svenska Föreningen för Grafisk Databehandling,
Gävle University,
VISKOM – Graphics and Visual Communication

Edited by

Stefan Seipel

Published for Svenska Föreningen för Grafisk
Databehandling (SIGRAD) by
Linköping University Electronic Press
Linköping, Sweden, 2004

Linköping Electronic Conference Proceedings, No. 13
Linköping University Electronic Press
Linköping, Sweden, 2004

ISBN 91-85295-97-3 (print)
ISSN 1650-3686 (print)
http://www.ep.liu.se/ecp/013/
ISSN 1650-3740 (online)

Print: UniTryck, Linköping, 2004

Table of Contents

Welcome to SIGRAD2004 in Gävle!

The annual Sigrad conference has established itself as *the* venue in Sweden for researchers and industry in the field of computer graphics, visual simulation and visualization. As in previous years, the Sigrad conference is held as a two-day event and is this year hosted by the University of Gävle. The special theme of SIGRAD2004 is *Environmental Visualization*. Two invited speakers and several paper contributions are aimed at this topic and are presented during day one of the conference. As is tradition, the scientific program is completed by quality contributions from other fields in computer graphics spanning both days of the event.

The goal of the SIGRAD2004 conference is to provide a Nordic (yet international) forum for communication of recent research and development results in computer graphics, visual simulation and visualization. SIGRAD2004 provides a forum for established researchers in the field to exchange their experiences with industry. SIGRAD2004 invites students and researchers in computer graphics to establish and deepen their academic networks as well as to foster new blood in the field.

During the past years computer graphics research has progressed dramatically and has created its own public image with impressive cinematic and game effects.
Likewise, computer graphics and visualization has evolved into powerful visual tools, which support us in daily work and decision-making. Visualizations of processes and information related to our environment have become ubiquitous though less spectacular. Urban and landscape planning, communication of weather forecasts and presentation of geographically related information are only few examples, where we visually explore environmental information.
We are proud to have Professor Oliver Deussen from the University of Constance as keynotes speaker at SIGRAD2004. He is one of the leading researchers in the field of vegetation simulation and landscape rendering. He is co-founder of Greenworks Organic Software, which produces xFrog, a highly successful software for procedural organic simulation. We are also pleased to have Dr. Tomas Landelius as invited speaker under the conference special theme. He is researcher at the Swedish Meteorological and Hydrological Institute and represents a field of expertise that uses various forms visualizations to convey complex information.Our third invited speaker is Professor Ken Museth at Linköpings University, campus Norrköping. In a session during the second day, he will present some of his latest research in the area of visualizing large unstructured datasets generated from computational fluids dynamics and structural mechanics simulations.

We would like to express our gratitude and warm welcome to the keynote speakers, authors of contributed papers, and other participants. We would also like to thank our sponsors, University of Gävle, Linköping University, The County Council of Gävleborg, EU Structural Funds, SIGRAD, Creative Media Lab, and the University of Linköping.

We wish you a most pleasant stay in Gävle.

Stefan Seipel
Chair, Program committee

Anders Backman
Chair, SIGRAD

SIGRAD2004 Program Committee

The SIGRAD2004 program committee consisted of experts in the field of computer graphics and visualization from all over Sweden. We thank them for their comments and reviews.

Dr. Mike Connell, CKK, Chalmers
Dr. Matthew Cooper, Linköping University, ITN
Dr. Mark Eric Dieckmann, Linköping University, ITN
Dr. Hans Frimmel, Uppsala University
Dr. Anders Hast, University of Gävle
Dr. Kai-Mikael Jää-Aro, KTH
Docent Lars Kjelldahl, KTH
Claude Lacoursière, Umeå University and CMLabs
Professor Haibo Li, Umeå University
Professor Ken Museth, Linköping University, ITN
Professor Stefan Seipel, Uppsala University and Gävle University

SIGRAD2004 Organizing Committee

Annika Erixån, University of Gävle
Ann-Kristin Forsberg, University of Gävle - VISKOM
Anders Hast, University of Gävle
Sharon Lazenby, University of Gävle
Roland Norgren, Creative Media Lab, University of Gävle
Daniel Wesslén, University of Gävle - VISKOM

SIGRAD Board for 2004

Anders Backman, Chair
Anders Hast, Member
Kenneth Holmlund, Substitute
Kai-Mikael Jää-Aro, Secretary
Lars Kjelldahl, Treasurer
Ken Museth, Substitute
Mark Ollila, Substitute
Stefan Seipel, Vice Chair
Odd Tullberg, Substitute
Örjan Vretblad, Substitute
Anders Ynnerman, Member
Charlotte Åkerlund, Substitute

Interactive landscape visualization

Oliver Deussen

University of Constance

Abstract

In recent years the development of graphics hardware and efficient rendering algorithms enabled researchers and game developers to create and render large landscapes with interactive rates. However, the shown scenes are still rough approximations that do not reach the complexity of real nature. To obtain sufficient simulations, a couple of problems have to be solved.

Creating a good scene requires powerful modelling algorithms on different levels. First a sufficient set of plant models has to be created. Nature is very diverse: modelling the most important plants that are found in Europe requires thousands of different models, and this is why efficient modelling algorithms for plants have to be found. In the talk I will present our xfrog modelling system, which was designed for that task.

The plant models then have to be combined to a virtual landscape. At this stage another modelling program is needed that enables the user to edit a huge number of small objects. Even a small garden consists of tens of thousands of plants, one square kilometre incorporates billions of single plants, and even storing the plant positions is here a problematic task. The plants interact with each other; complex patterns arise due to seeding mechanisms and the fight for resources. Sometimes, the development of a landscape has to be simulated.

Having modelled plant models and positions, we end up with tons of geometry even for a small landscape. A single tree model consists sometimes of millions of surfaces, a forest of billions. Efficient level-of-detail algorithms are necessary to obtain interactive rendering with these scenes. This can be done by representing the plant models by billboards or point clouds. The size of the representation is computed for each model and frame and thus allows us to carefully adapting the shown geometry to what is necessary.

Rendering the models is also an interesting problem. The interaction of light with the plant surfaces and especially leaves is not trivial. Subsurface scattering and different optical properties of plant tissues makes it necessary to adapt standard rendering techniques to these models. Especially for hardware rendering this is a complex task.

In the talk I will outline our modelling and rendering pipeline and show some of the algorithms we implemented. Also I will review the problems and frontiers we currently focus while solving our goal of rendering one square kilometre of nature at interactive frame rates.

Speaker Bio

Oliver Deussen graduated in 1996 at the University of Karlsruhe about graphical simulation techniques. From 1996 to 2000 he was research assistant at the Department of Simulation and Computer Graphics at the University of Magdeburg.
In 2000 he received an associate professorship at the Dresden University of Technology. Since 2003 he is full professor at the University of Constance and chair head for computer graphics and media informatics. His research topics include modelling and visualization of complex landscapes, non-photorealistic rendering and information visualization.

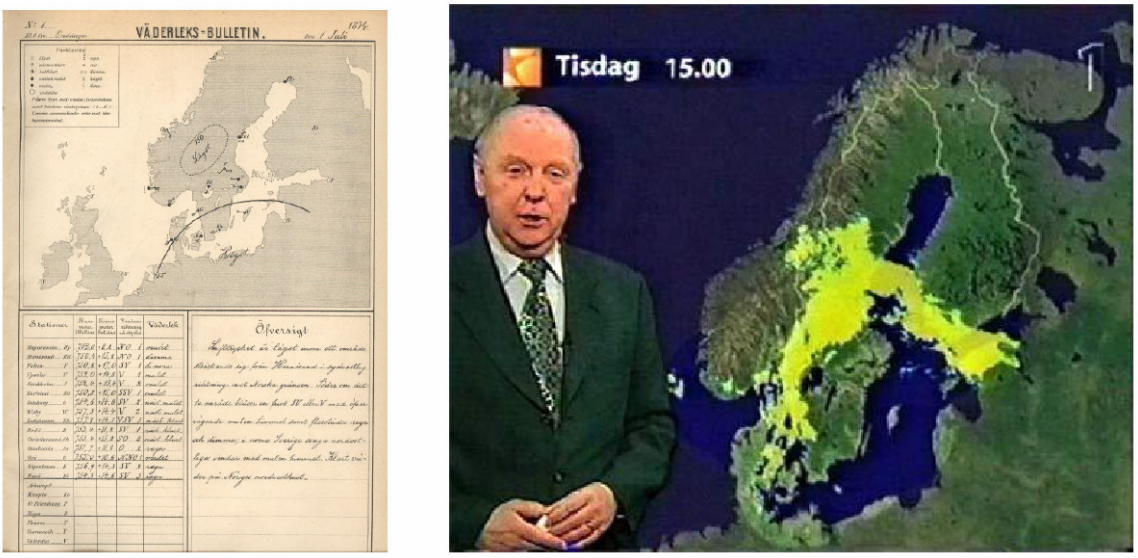
Visualization at SMHI

Tomas Landelius

Swedish Meteorological and Hydrological Institute
SE 601 76 Norrköping

Abstract

Like most other national weather services the Swedish Meteorological and Hydrological Institute (SMHI) was established in the late 19th century. The reason for this was the development of the telegraph. In order to make a weather forecast one need information about the current weather situation within a large area and the telegraph made this possible. As an illustration, todays 24 hour weather forecasts for Sweden use information from all of Europe, the northern Atlantic as well as the eastern parts of northern America.
Accordingly SMHI gathers vast quantities of data around the clock from land-based weather stations, balloons, ships, buoys, aircraft, weather radar, satellites and lightning localization systems. This information is the base for further processing in complex numerical models, tailored for applications in meteorology, hydrology and oceanography (MHO).



Left: First public weather bulletin from SMHI in 1874. Right: Modern TV weather presentation with superimposed radar image sequence.

SMHI operates under the auspices of the Swedish Ministry of the Environment and uses its MHO expertise to promote efficiency, safety and a better environment. In order to do this forecasts, severe weather warnings and other model results need to be analyses, presented and visualized in ways that meet the specific needs of a wide spectra of audiences. These range from researches, developers and forecaster to customers in various areas of society.
Weather information to the public has for a long time been an important task as illustrated in the figure above. This talk will serve a smorgasbord with several examples of how visualization techniques are used at SMHI. It will become evident that although the processes in the atmosphere and ocean take place in four dimensions almost all visualizations at SMHI are done in 2-D. Some possible reasons why this is the case will be discussed in the talk.

Speaker Bio

Tomas Landelius is a research scientist at the Atmospheric research group at SMHI where he applies his knowledge in multidimensional signal analysis, non-linear regression, optimal control and optimization to a diversity of problems: Assimilation of satellite micro wave data in weather prediction models (NWP), analysis of satellite cloud images and radar imagery as well as the development of solar radiation models for the UV, visible and near-infrared.
His graduate work concerned the development of a novel algorithm for reinforcement learning in high-dimensional signal spaces and he also made contributions concerning the use of canonical correlation in image analysis applications. He received his M.Sc. in Computer Science and Technology (1990) as well as his Ph.D. in Computer Vision (1997) from Linköping University, Sweden.

TetSplat: Real-Time Rendering and Volume Clipping of Large Unstructured Tetrahedral Meshes

Ken Museth

Linköping University

Abstract

Museth will present a novel approach to interactive visualization and exploration of large unstructured tetrahedral meshes. These massive 3D meshes are used in mission-critical CFD and structural mechanics simulations, and typically sample multiple field values on several millions of unstructured grid points. Our method relies on the preprocessing of the tetrahedral mesh to partition it into non-convex boundaries and internal fragments that are subsequently encoded into compressed multi-resolution data representations. These compact hierarchical data structures are then adaptively rendered and probed in real-time on a commodity PC. Our point-based rendering algorithm, which is inspired by QSplat, employs a simple but highly efficient splatting technique that guarantees interactive frame-rates regardless of the size of the input mesh and the available rendering hardware. It furthermore allows for real-time probing of the volumetric data-set through constructive solid geometry operations as well as interactive editing of color transfer functions for an arbitrary number of field values.
Thus, the presented visualization technique allows end-users for the first time to interactively render and explore very large unstructured tetrahedral meshes on relatively inexpensive hardware.

Further information on the research group can be found at www.gg.itn.liu.se and a paper presenting the current work is available at www.gg.itn.liu.se/Publications/Vis04.

Incremental Spherical Linear Interpolation

Tony Barrera *
Barrera Kristiansen AB

Anders Hast †
Creative Media Lab,
University of Gävle

Ewert Bengtsson ‡
Centre for Image Analysis
Uppsala University

Abstract

Animation is often done by setting up a sequence of key orientations, represented by quaternions. The in between orientations are obtained by spherical linear interpolation (SLERP) of the quaternions, which then can be used to rotate the objects. However, SLERP involves the computation of trigonometric functions, which are computationally expensive. Since it is often required that the angle between each quaternion should be the same, we propose that incremental SLERP is used instead. In this paper we demonstrate five different methods for incremental SLERP, whereof one is new, and their pros and cons are discussed.

1 Introduction

We propose that two approaches previously published for spherical linear interpolation (SLERP) [Shoemake 1985] of vectors and intensities [Barrera 2004; Hast 2003], also are used for SLERP of quaternions, a technique often used in animation. We also demonstrate two other ways of performing incremental SLERP by using quaternion multiplication and by using matrices directly. Furthermore we propose a new way of doing incremental SLERP by posing the problem as solving a differential equation using the Euler method.

1.1 Quaternions and Animation

Animation is often done by setting up a sequence of key orientations, represented by quaternions. The in between orientations are obtained by spherical linear interpolation of the quaternions, which then can be used in order to rotate the objects [Svarovsky 2000]

$$p' = qpq^{-1} \tag{1}$$

where q is a unit quaterion and p is a position i 3D in quaternion form

$$p = (x, y, z, 0) \tag{2}$$

Unit quaternions are simply another representation of rotation matrices. It is possible to convert quaternions into matrices and vice versa. Actually it can be proven that a unit quaternion will be the eigenvector of the corresponding orthonormal rotation matrix [Kuipers 1999]. The quaternion consists of four elements. The first three are the sine of half the rotation angle multiplied by a unit vector **n** that the rotation is performed around. The last one is the cosine of half the rotation angle

$$q = (\mathbf{v}, s) = (\mathbf{n}\sin(\theta/2), \cos(\theta/2)) \tag{3}$$

For a unit quaternion, the conjugate is the same as the inverse, which is defined as

$$q^{-1} = (-\mathbf{v}, s) \tag{4}$$

1.2 SLERP

The theory behind SLERP, was introduced to the computer graphics society by Shoemake [Shoemake 1985] and an interesting proof is given by Glassner [Glassner 1999]. SLERP is different from linear interpolation in the way that the angle between each quaternion will be constant, i.e. the movement will have constant speed. Linear interpolation (lerp) will yield larger angles in the middle of the interpolation sequence. This will cause animated movements to accelerate in the middle, which is undesirable [Parent 2002]. Moreover, will quaternions interpolated by lerp be shorter and normalization is therefore necessary. However, quaternions obtained by SLERP do not need to be normalized, since SLERP will always yield unit quaternions as long as the quaternions being interpolated in between are unit quaternions.

The formula used is

$$q(t) = q_1 \frac{\sin((1-t)\theta)}{\sin(\theta)} + q_2 \frac{\sin(t\theta)}{\sin(\theta)} \tag{5}$$

where $t \in [0,1]$, and θ is the angle between q_1 and q_2 computed as

$$\theta = cos^{-1}(q_1 \cdot q_2) \tag{6}$$

2 Fast incremental SLERP

It is shown in [Hast 2003] that equation (5) can be rewritten as

$$q(n) = q_1 \cos(nK_\theta) + q_o \sin(nK_\theta) \tag{7}$$

where q_o is the quaternion obtained by applying one step of Gram-Schmidt's orthogonalization algorithm [Nicholson 1995] and then it is normalized. This quaternion is orthogonal to q_1 and lies in the same hyper plane spanned by q_1 and q_2. Furthermore, if there are k steps then the angle between each quaternion is $K_\theta = \frac{\cos^{-1}(q_1 \cdot q_2)}{k}$. In the subsequent matlab code examples, it is assumed that q_1 and q_2 are unit quaternions and that the following five lines are included in the beginning of the code

```
qo=q2-(dot(q2,q1))*q1;
qo=qo/norm(qo);
t=acos(dot(q1,q2));
kt=t/k;
q(1,:)=q1;
```

*e-mail: tony.barrera@spray.se
†e-mail: aht@hig.se
‡e-mail: ewert@cb.uu.se

The code that follows on these three lines can be used for computing incremental SLERP. Usually SLERP is computed using trigonometric functions in the inner loop in the following way

```
b=kt;
for n=2:k+1
  q(n,:)=q1*cos(b)+qo*sin(b);
  b=b+kt;
end
```

The cost for computing SLERP in this way is one scalar addition, one sine and one cosine evaluation, two scalar-quaternion multiplications and one quaternion addition. This is not very efficient and the following subsections will show how these trigonometric functions can be avoided.

2.1 De Moivre

In [Hast 2003] it is shown that complex multiplication can be used in order to compute SLERP efficiently. Hence, the computationally expensive trigonometric functions are avoided.

Complex numbers are defined in an orthonormal system where the basis vectors are $\mathbf{e_1} = 1$ and $\mathbf{e_2} = i$. The De Moivre's formula [Marsden 1996] states that

$$(\cos(\theta) + i\sin(\theta))^n = \cos(n\theta) + i\sin(n\theta) \tag{8}$$

The right part of the formula looks very similar to equation (7). In fact is possible to compute the left part instead, by using one complex multiplication. However, there are still some things to arrange before we can compute SLERP of quaternions instead of complex numbers.

Let Z be a complex number computed by

$$Z = \cos(K_\theta) + i\sin(K_\theta) \tag{9}$$

Furthermore, we treat complex numbers as if they were vectors in 2D-space, by defining a product which is similar to the ordinary dot product

$$(q_1, q_o) \bullet (\Re(Z), \Im(Z)) = q_1\Re(Z) + q_o\Im(Z) \tag{10}$$

Thus, we compute the SLERP as

$$q(n) = (q_1, q_o) \bullet Z^n \tag{11}$$

Finally, we prove that this will yield the desired result. Rewrite equation (11) by using equation (8)

$$\begin{aligned} q(n) &= (q_1, q_o) \bullet Z^n \\ &= (q_1, q_o) \bullet (\cos(n\theta) + i\sin(n\theta)) \end{aligned} \tag{12}$$

Then, expand this equation by using equation (10)

$$q(n) = q_1\cos(nK_\theta) + q_o\sin(nK_\theta) \tag{13}$$

The matlab code for this approach is

```
Z1=cos(kt)+i*sin(kt);
Z=Z1;
for n=2:k+1
  q(n,:)=q1*real(Z)+qo*imag(Z);
  Z=Z1*Z;
end
```

The cost for computing the intensity will be one complex multiplication and one two scalar-quaternion multiplications and one quaternion addition.

2.2 Chebyshev

In [Barrera 2004] a faster approach is derived using the Chebyshev recurrence [Burden 2001]. This will make the computation in the inner loop much more effective than using the complex multiplication.

The Chebyshev's recurrence relation is

$$T_{n+1}(u) = 2uT_n(u) - T_{n-1}(u) \tag{14}$$

where $T_n(u)$ are Chebyshev polynomials [Burden 2001] of the first kind that can be written as

$$T_n(u) = \cos(n\phi) = \cos(n\cos^{-1}u) \tag{15}$$

In order to obtain the Chebyshev recurrence for solving equation (7) let

$$u = \cos(K_\theta) \tag{16}$$

$$\phi = K_\theta \tag{17}$$

Then equation (15) gives

$$\cos(nK_\theta) = \cos(n\cos^{-1}u) \tag{18}$$

This equation can be solved by equation (14). Since sin is just a phase shifted cos it will also be solved by the proposed equation as long as the angle is the same for both. Let

$$q(n) = q_1\cos(nK_\theta) + q_o\sin(nK_\theta) \tag{19}$$

For the setup starting with $n = 0$, we subsequently have

$$\begin{aligned} q_{-1} &= q_1\cos(-K_\theta) + q_o\sin(-K_\theta) \\ &= q_1\cos(K_\theta) - q_o\sin(K_\theta) \end{aligned} \tag{20}$$

and

$$\begin{aligned} q_0 &= q_1\cos(0) + q_o\sin(0) \\ &= q_1 \end{aligned} \tag{21}$$

Now, we can put together our algorithm. We can also optimize it somewhat by removing the factor 2 from the loop

```
tm1=q1*cos(kt)-qo*sin(kt);
t0=q1;
u=2*cos(kt);
for n=2:k+1
  tp1=u*t0-tm1;
  q(n,:)=tp1;
  tm1=t0;
  t0=tp1;
end
```

The cost for this approach is one scalar-quaternion multiplication, one quaternion subtraction and two quaternion moves.

2.3 Quaternion Power Function

The power function variant of SLERP for quaternions [Shankel 2000]can also be used for incremental SLERP. It is defined as

$$q(n) = q_1(q_1^{-1}q_a)^n \tag{22}$$

We have to compute q_a in such a way that we will obtain intermediate quaternions in between q_1 and q_2

$$q_a = q_1\cos(K_\theta) + q_o\sin(K_\theta) \tag{23}$$

then we obtain the following algorithm

```
q1i=[-q1(1), -q1(2), -q1(3), q1(4)];
qa=q1*cos(kt)+qo*sin(kt);
qb=qmul(q1i,qa);
q0=q1;
for n=2:k+1
    q0=qmul(q0,qb);
    q(n,:)=q0;
end
```

Here *qmul* is an implementation of quaternion multiplication, which is defined as

$$q_1 q_2 = (\mathbf{v}_1, s_1)(\mathbf{v}_2, s_2) \tag{24}$$
$$= (s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2, s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2) \tag{25}$$

The cost for this approach is only one quaternion multiplication in the inner loop.

2.4 Euler

Here we introduce how it is possible to perform incremental SLERP by solving a ordinary differential equation using the Euler method [Gerald 1994]. It is easy to show that

$$y = y_1 \cos(K_\theta) + y_o \sin(K_\theta) \tag{26}$$

is a solution to the ordinary differential equation [Simmons 1991]

$$y'' = -k^2 y \tag{27}$$

where k is a constant. The Euler method gives an approximation to the solution by the following forward difference

$$y_{i+1} = y_i + y' \tag{28}$$

However, in order to make this an exact computation we shall compute the derivate using Eulers method as well

$$y'_{i+1} = y'_i + y'' \tag{29}$$

The trick is to use the fact that $y'' = -k^2 y$, then we can nest the recurrence as

$$y'_{i+1} = y'_i - k^2 y_i \tag{30}$$
$$y_{i+1} = y_i + y'_{i+1} \tag{31}$$

Substitute equation (30) into (31)

$$y_{i+1} = y_i + y'_i - k^2 y_i \tag{32}$$

Now, we make use of the fact that $y'_i = y_i - y_{i-1}$ and put that into equation (32). Thus

$$y_{i+1} = y_i + y_i - y_{i-1} - k^2 y_i \tag{33}$$

After rearranging the terms we get

$$y_{i+1} = (2 - k^2) y_i - y_{i-1} \tag{34}$$

If we let $(2 - k^2) = 2u$, where $u = \cos(K_\theta)$ then equation (34) is the same as the Chebyshev recurrence (14). This implies that the nested Euler method can be used to compute incremental SLERP exactly. Therefore

$$k^2 = 2 - 2\cos(K_\theta) \tag{35}$$

In this case we have

$$y_{-1} = y_1 \cos(K_\theta) - y_o \sin(K_\theta) \tag{36}$$
$$y_0 = y_1 \tag{37}$$
$$\tag{38}$$

And

$$y'_0 = y_0 - y_{-1} \tag{39}$$
$$= y_1 - y_1 \cos(K_\theta) + y_o \sin(K_\theta) \tag{40}$$
$$= y_1 (1 - \cos(K_\theta)) + y_o \sin(K_\theta) \tag{41}$$

Putting it all together yields the following algorithm

```
p=1-cos(kt);
y=q1;
k2=2*p;
y1=qo*sin(kt)+p*q1;

for n=2:k+1
    y1=y1-k2*y;
    y=y+y1;
    q(n,:)=y;
end
```

The cost is one scalar-quaternion multiplication, one quaternion subtraction and one quaternion addition.

2.5 Matrix Multiplication

The previously presented algorithms can be useful if quaternions are used to perform the actual rotation of the objects. However, if the quaternions must be transformed into matrices it is better to do the interpolation using matrices all along.

Let

$$M_1 = quattomat(q_1) \tag{42}$$
$$M_a = quattomat(q_1 \cos(K_\theta) + q_o \sin(K_\theta)) \tag{43}$$

where *quattomat* [Watt 1992] is a function that transforms a quaternion into a matrix. The there must exist a matrix, M that transforms M_1 into M_a

$$M_1 M = M_a \tag{44}$$

Thus

$$M = M_1^{-1} M_a \tag{45}$$

The code is

```
M1=quattomat(q1);
Ma=quattomat(q1*cos(kt)+qo*sin(kt));
M=inv(M1)*Ma;
Q=M1;
for n=2:k+1
    Q=Q*M;
end
```

3 Conclusions

It is possible to evaluate SLERP for quaternions used in animation in a very efficient way using incremental SLERP instead of evaluating the original SLERP functions containing trigonometric functions in the loop.

All presented algorithms will yield the same intermediate quaternions, or as in the case for the matrix version, the corresponding rotation matrix. They are all interpolation approaches and not approximations. The only error introduced is by the floating point arithmetic itself. They are all therefore numerically stable algorithms.

The fastest approach for quaternion SLERP is the one derived from Chebyshev's recurrence relation. The others may seem redundant. However, the purpose of this paper is to show that there

are many ways to avoid the computationally expensive trigonometric functions in the inner loop. Perhaps the other approaches might have other uses in other contexts. The algorithm derived from De Moivre's formula might turn out to be useful if complex multiplication is implemented in hardware. The same goes for the Quaternion power function approach. The one using the Euler method is interesting as an alternative to the Chebyshev approach since the variable swap is replaced by a quaternion addition.

If the rotation is done by matrix multiplication instead of quaternion rotation, then the Matrix multiplication approach can be even faster than the Chebyshev approach since matrix multiplication often is implemented in hardware and quaternion multiplication (for the rotation of the objects) is usually not. This of course heavily depend of which platform is being used for animation.

Anyhow, many graphics applications and especially animation could benefit from having quaternion arithmetics implemented in hardware. That would make incremental SLERP very fast. Nonetheless, a software implementation using any of the proposed approaches for SLERP will still be much faster than using trigonometric functions in the inner loop.

References

T. BARRERA, A. HAST, E. BENGTSSON 2004. *Faster shading by equal angle interpolation of vectors* IEEE Transactions on Visualization and Computer Graphics, pp. 217-223.

R. L. BURDEN, J. D. FAIRES 2001. *Numerical Analysis* Brooks/Cole, Thomson Learning, pp. 507-516.

C. F. GERALD, P. O. WHEATLEY 1994. *Applied Numerical Analysis, 5:th ed.* Addison Wesley, pp. 400-403.

A. GLASSNER 1999. *Situation Normal* Andrew Glassner's Notebook- Recreational Computer Graphics, Morgan Kaufmann Publishers, pp. 87-97.

A. HAST, T. BARRERA, E. BENGTSSON 2003. *Shading by Spherical Linear Interpolation using De Moivre's Formula* WSCG'03, Short Paper, pp. 57-60.

J. B. KUIPERS 1999. *Quaternions and rotation Sequences - A Primer with Applications to Orbits, Aerospace, and Virtual Reality* Princeton University Press, pp. 54-57, 162,163.

J. E. MARSDEN, M. J. HOFFMAN 1996. *Basic Complex Analysis* W. H. Freeman and Company, pp. 17.

W. K. NICHOLSON 1995. *Linear Algebra with Applications* PWS Publishing Company, pp. 275,276.

R. PARENT 2002. *Computer Animation - Algorithms and Techniques* Academic Press, pp. 97,98.

J. SHANKEL 2000. *Interpolating Quaternions* Game Programming Gems. Edited by M. DeLoura. Charles River Media, pp. 205-213

K. SHOEMAKE 1985. *Animating rotation with quaternion curves* ACM SIGGRAPH, pp. 245-254.

G. F. SIMMONS 1991. *Differential Equations with Applications and Historical Notes, 2:nd ed.* MacGraw Hill, pp. 64,65.

J. SVAROVSKY 2000. *Quaternions for Game Programming* Game Programming Gems. Edited by M. DeLoura. Charles River Media, pp. 195-299.

A. WATT, M. WATT 1992. *Advanced Animation and Rendering Techniques - Theory and Practice* Addison Wesley, pp. 363.

Dynamic Code Generation for Realtime Shaders

Niklas Folkegård*
Högskolan i Gävle

Daniel Wesslén†
Högskolan i Gävle

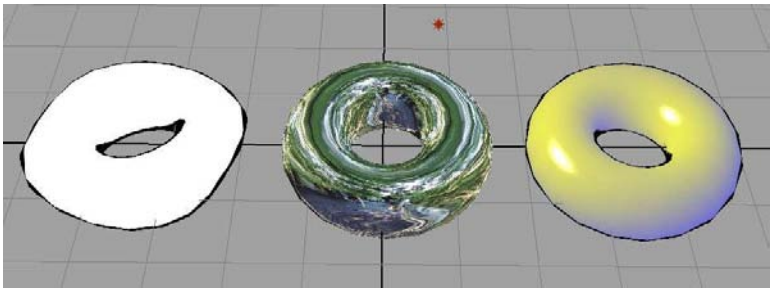


Figure 1: Tori shaded with output from the Shader Combiner system in Maya

Abstract

Programming real time graphics has been made more efficient by introducing high level shading languages such as Cg and GLSL. Writing shader programs can be redundant since the same functions appear in many shaders. This article suggests a method to combine single functions and create compound shaders in runtime. Redundancy is avoided by dividing programs into smaller, reusable parts. An algorithm is presented for joining these parts into working shaders based on just a few parameters.

CR Categories: I.3.0 [Computer Graphics]: General— [I.4.8]: IMAGE PROCESSING—Shading D.1.2 [Software]: Programming Techniques—Automatic Programming

Keywords: Graphics hardware, GLSL, GPU programming, programming efficiency, real time shaders, dynamic code generation, meta programming, computer graphics

1 Introduction

Real time graphics have become significantly faster and better with the introduction of programmable graphics cards (GPU:s) for the consumer market. The first of these GPU:s were released in 2001 [Fernando and Kilgard 2003]. Since these only could be programmed on an assembly level, the work of creating faster real time graphics became a lengthy and expensive procedure. In 2002, Nvidia Corporation released a high level shader language for programming this type of hardware, Cg (C for Graphics) [Mark et al. 2003], and during 2003 this was followed by the release of The OpenGL Shading Language, GLSL [Kessenich et al.].

*e-mail: nfd@hig.se
†e-mail: dwn@hig.se

With the introduction of high level shader languages, the work of creating high end real time graphics has become faster, easier and more comprehensible. But for large scale projects, where many different shaders are used, the process of writing shader programs still takes an unnecessary amount of time. The main reason for this is redundancy — even though many shaders share the same concepts, they still have to be written into each individual program. It would be preferrable if each concept only had to be written once, thus making the choice of concepts the relevant task in shader creation.

This article aims to present a solution where a complete shader program is created dynamically from short sections of code, representing the various concepts of shading algorithms. Hopefully, this solution will make using shader languages for implementation of hardware based real time graphics simpler and more efficient.

2 Background

2.1 Shader Languages

Most shader languages resemble C, but are specialized for simple computations involving vectors and matrices. The languages address the construction of the graphics hardware, where each vertex is sent through a pipeline where the position and material properties of the vertex are extracted and transformed to be used in calculating the colour of the fragments. Therefore the shader languages can affect the look of the graphics in two steps of the computations: before the vertex is transformed into fragments, and before the fragment is transformed into a pixel [Fernando and Kilgard 2003; Rost 2004].

Shader languages support run-time compilation. The application using shader languages as a support system for graphics computations can store the code as strings and call a built-in compiler when the program is needed. In most cases, just one program of each type may be on the GPU, but the application can easily swap shader programs in run-time [Fernando and Kilgard 2003; Rost 2004].

2.2 Automatic Creation of Shaders

One of the many benefits of shader languages like GLSL and Cg is that they make the borders between CPU and GPU based compu-

tations clear. In order to uphold this clarity some traditional methods for reducing code redundancy, such as procedural abstraction, must be omitted. Each shader program must be sent to the GPU as one unit, and as soon as it has started there is no good way to return to the CPU to call external functions. Because of these limitations, shader programs have typically been written as one big main-function where common concepts have been manually entered wherever they have been needed, thus bringing redundancy to any project of size.

Adding to the complexity of the problem, there is also the issue of combinatorics. Recent research has shown that it is possible to find faster computations for some cases of shading [Hast 2004]. This brings a need for a system able to handle many implementations of the same concept, and the ability to choose which implementation to use based on the properties of the rendering task at hand. Adding this kind of flexibility to a project with all hand-written shader programs would make it necessary to write duplicates of each shader using these concepts. The work load could easily become overwhelming when the number of alternative implementations increase.

There are ways to overcome the problem, however. By dividing the code of shader programs into smaller parts and providing some means by which these parts can be joined to form new programs, the work of creating shader effects can be made less redundant. Concepts can easily be defined in one place, and reused in many. While there are some implementations of this idea [Bleiweiss and Preetham 2003; LightWorks 20/05/2004; McCool et al. 2004; RTzen inc.], the solutions available do not fully encompass the following goals:

- code redundancy reduction
- ability to choose from many different implementations of the same concept
- ability to automatically include necessary, intermediate code
- clear borders between CPU and GPU

The solution presented in this article will show a way to overcome some of the limitations of established shader languages, without inventing a new language or restricting the artist to work with predefined algorithms.

3 Shader Combiner

This section will present a system that, based on a few parameters, automatically links parts of shader code together to form working shader programs. Manually created by the programmer, these shader code parts should be made globally available in the system. By specifying the criteria for a particular shader a working solution should be constructed by the system. If there are many solutions to the same query it should be possible to choose which one of the solutions to use. The implementation which this article is based on is made using GLSL, but the ideas expressed can easily be transferred to any shading language with similar properties.

3.1 Dividing Code into Parts

A shader program can be seen as consisting of operations that need to be performed in a certain order. Some operations set the value of certain variables, while others use these variables as part of their computations. Looking at shader programs this way, it becomes obvious that a program can be divided into parts as long as we can make sure that these parts appear in the right order whenever they are rejoined to form a new program. To ensure this it is necessary to provide each part with some properties that can be used for linking it to other parts. Taking into account that each part actually does something useful as well, we end up with three important properties that define a part:

1. inputs, or preconditions, representing the concepts needed by a part
2. computations, or operations, representing the actual work of a part
3. outputs, or postconditions, representing the result of the computations that might be useful for other parts

To add flexibility and ease of use, the implementation uses a larger set of properties for its parts, but these three properties are all that are conceptually needed. From this description it is possible to create parts representing the concepts needed for shading.

3.2 Combination Complexity

Today's programmable GPU's afford operation control in two stages of computations — in vertex computations and in fragment computations. Thus a solution could mean that two separate programs should be used, one in each stage. Consequently it is useful to consider a solution as being a pair of programs, where both parts of the pair should be used to reach the desired result. The system needs to make sure that two separate programs work without conflicting with each other, and it needs to do this even when none of the two programs is fully constructed.

A part can provide any number of outputs and require any number of inputs, a fact that could slow down the process of construction. In a small project there might be just one part for each concept, making the work of combining the parts easy. For larger projects, however, there could be many different parts that all need to be considered by the system. The many parts and their many connections can also form circular dependencies, where part *A* needs input from part *B*, while *B* needs input from *A*, thus making an infinite loop.

3.3 Rejoining Code Parts

To reach a solution the user should state the conditions to be met by the program pair. The system will then search through the available parts and return a program pair fulfilling the conditions stated. In order to achieve this we need an algorithm $getShader(C)$ that from a set of conditions C will return nought or one working program pair. This section will present such an algorithm.

3.3.1 Solution parts

The operations in each part are specific for a certain state in the programmable graphics pipeline. A fragment part can only fulfill conditions for other fragment parts, while a vertex part can fulfill conditions for other vertex parts as well as for fragment parts. In the future other parts of the pipeline are likely to be accessible for programming and parts of a future type will thus have their own rules for how they affect other types in the system. However, to simplify this explanation we just note that there is a difference between how different types of operations affect each other, and assume that the system can handle these rules in a way invisible to the user.

The user can provide the system with solution parts as he/she finds necessary, and these parts will be used by the system. Note that the conditions are global, e.g. if a part P_1 has a precondition called *normal* and a part P_2 has a precondition that is also called *normal* they will both benefit from a part P_3 having a postcondition called *normal*. Therefore the parts can be connected to each other, so that one or more parts fulfill all preconditions of another part. A part having no preconditions is complete and will need no more help from the other parts in the system. Listing 1 shows a selection of solution parts from the implementation.

Listing 1: Solution parts

```
#vertex 10 "stdVertex" manual
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
        out vec3 viewer=normalize(vec3(gl_ModelViewMatrix * gl_Vertex));
        out vec3 vNormal = normalize(gl_NormalMatrix * gl_Normal);

#vertex 10 "stdDiffuse" manual
        out vec3 diffuseMat =  gl_FrontMaterial.diffuse.rgb;

#vertex 10 "stdLight0" manual
        out vec3 light0Pos = gl_LightSource[0].position.xyz;
        out vec3 diffuseLight = gl_LightSource[0].diffuse.rgb;

#vertex 10
        in vec3 viewer;
        in vec3 light0Pos;
        out vec3 vLight0 = normalize(light0Pos - viewer);

#vertex 10
        in vec3 vLight0;
        in vec3 vNormal;
        out float vNdotL = dot(vLight0 , vNormal);

#vertex 10 "stdLambert" manual
        in float vNdotL;
        in vec3 diffuseMat;
        in vec3 diffuseLight;
        out vec3 lambert = vec3(diffuseMat * max(0.0, vNdotL) *
                diffuseLight);

#fragment 100 "fLambertOut"
        in vec3 lambert;
        gl_FragColor = vec4(lambert, 1.0);
```

3.3.2 Representation of the problem

Knowing the structure of the solution parts, it is now possible to represent the shader program requested by the user. If we take a solution part P and let its preconditions consist of the conditions C specified by the user (the caller of the algorithm $getShader(C)$), and in all other aspects leave this part empty, we will have a solution part demanding the functions specified by the user but not doing anything in itself. If we perform a search through the system using this part as a starting point, it will be connected with other parts fulfilling the preconditions. When the part is complete we have a solution.

In reality it is not all that simple. As mentioned earlier today's hardware have two programmable stages — the vertex stage and the fragment stage — and since we do not know which type of solution part will fulfill certain preconditions there is always a risk that the system returns two separate programs. Moreover, these programs must work together in an optimal way (for example, as many operations as possible should be performed in the vertex stage since this stage is used less frequently). Since the desired solution may be a pair of vertex and fragment programs, the problem will throughout this explanation be called a *program pair*. We also note that, like the solution parts, a program pair having no preconditions is complete.

3.3.3 Linking parts into a solution

Initially, a program pair is created whose preconditions correspond to the user's demands. The system can now search for solution parts that fulfill these preconditions. If it is possible to connect this program pair with solution parts fulfilling all initial preconditions we have a *possible* solution, but new preconditions may arise from the parts found and it is not until all preconditions are fulfilled that we have a complete solution.

The parts that are found helpful are connected to the program pair by letting the preconditions of each helpful part P become part of the preconditions of the program pair. At the same time, the postconditions of each part P eliminate preconditions of the program pair. By this operation we get a new set of preconditions affecting how future search is done. If the set of preconditions is empty, we consider the program pair complete.

3.3.4 Finding possible solutions

Finding suitable solution parts should be fast and easy. Each time a solution part is found, new preconditions might be added, and each time multiple parts fulfill the same precondition the search path would branch. The best thing would be if we quickly could find a set of solution parts fulfilling all preconditions that are currently unfulfilled. This could be done by mapping each available postcondition in the system to the corresponding parts.

If the system only has one solution part for each entry in such a map, a list of preconditions could easily be converted to a list of matching solution parts. But the system could have mapped any number of parts in each entry. We need to find the different combinations of solution parts fulfilling all current preconditions. The internal order of the elements is of no importance. In this stage we are only concerned with whether a part is *useful*, not *in which order* it will be used.

If we do not find a set of solution parts fulfilling all preconditions of the program pair, we will not find a solution. If we find many sets, the search will branch.

3.3.5 Putting operations together into working programs

The program pair is built as a collection of solution parts needed to make it complete. As soon as it is, we need to transform this collection into working shader code. Each part has an element representing its operations, and this is now used to build the final shader programs. From the collection, all parts having no preconditions are added. The added parts provide some postconditions and gives us a new state. Now we can add all collected parts whose preconditions are fulfilled by this new state. This process goes on until the collection is emptied.

Note that this method will not be able to put together parts with circular dependencies. This behavior is desired, since such dependencies mean that the code is inoperable. If we in any step cannot put in new code we have a case where the parts cannot be joined into working code, and the current solution will fail. In this stage it is also possible to perform other tests on the code (e.g. we might want to try and compile the code, or test it against the state of the calling application).

If the search has branched, we will end up with many program pairs. What remains is selecting one of these solutions. Here it is possible for the user to decide the criteria to be used in the selection process. If no criteria are provided, the system will pick whichever solution it regards as the fastest, based on the sum of part costs.

3.3.6 The final algorithm

The algorithm initially sought for, $getShader(C)$, is now easy to define (see Algorithm 1). It will search the system, generate as many program pairs as possible and choose one of them to return.

Algorithm 1 $getShader(C)$

procedure GETSHADER(C)
  Construct a program pair α_0 whose preconditions are C
  for each α_n in the system **do**
    while α_n has preconditions **do**
      Find all sets of solution parts whose postconditions fulfill all preconditions of α_n
      for each set found **do**
        Add a new program pair α_m to the system
        Eliminate all preconditions of α_m
        Add the parts in the set to α_m
        Add the preconditions of all parts in the set to α_m
      end for
    end while
  end for
  for each α_n in the system **do**
    while α_n has parts **do**
      Mark all added parts whose preconditions are fulfilled by the postconditions of α_n
      Add the postconditions of all marked parts to α_n
      Add the code of all marked parts to α_n
      Eliminate all marked parts from α_n
    end while
  end for
  return the best α_n generated
end procedure

3.3.7 Public and explicit parts

In the description above, a global set of solution parts is assumed for the search. Cases may arise where some solution parts are not useful, even though they provide functionality demanded by the user. For instance, all variables built into GLSL assume a standardized format, and each solution part providing functionality by using these variables belong to a set of solution parts not suitable for global accessibility since we cannot assume that all objects rendered are standardized.

The solution to this problem is simple. In the previous we have assumed that searching is done in a global set. Now this set is divided into a *public* set consisting of parts that can always be used, and a *private* set consisting of solution parts that can only be used when they are explicitly asked for. Searching will thus be done in the public set and the set of parts explicitly asked for.

It is also possible that a user needs certain parts to be included in a solution. This too is easily attended to. We simply connect these parts to the initial program pair before starting the search. That way the preconditions will be updated to represent a stage where the demanded parts are included.

3.4 Implementation

3.4.1 Description

The implementation [Folkegard 03/06/2004] is GLSL specific, which most of all is seen in how it handles types and modifiers of global variables. In the algorithm, handling global variables has been avoided, but in the implementation it is an important part. Values assigned through *const* or *uniform* do not require operations in the shader program, and are therefore regarded as the fastest method. Vertex shader values used in a fragment shader must be passed by declaring *varying* variables. The implementation takes care of this handling.

The user writes the code for the parts in a simple syntax where preconditions, global variables, operations and postconditions are clearly indicated. The parts are then added to the system and stored in structures supporting fast finding. Searching is done by specifying three parameters (a,b,c). The parameters are lists of requested functions (a), explicitly demanded parts (b), and parts that should be made available in searching (c). The system can return strings with the code found by the system, as well as lists of global variables. Output from the implementation can be found in listings 2 and 3.

Listing 2: Simple Lambert Vertex Shader Automatically Created

```
varying vec3 lambert ;
void main()
{
        vec3 diffuseMat;
                diffuseMat = gl_FrontMaterial.diffuse.rgb ;
        vec3 vNormal;
        vec3 viewer;
                gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
                viewer=normalize(vec3(gl_ModelViewMatrix * gl_Vertex));
                vNormal = normalize(gl_NormalMatrix * gl_Normal) ;
        vec3 diffuseLight;
        vec3 light0Pos;
                light0Pos = gl_LightSource[0].position.xyz ;
                diffuseLight = gl_LightSource[0].diffuse.rgb ;
        vec3 vLight0;
                vLight0 = normalize(light0Pos - viewer) ;
        float vNdotL;
                vNdotL = dot(vLight0 , vNormal) ;

        lambert = vec3(diffuseMat * max(0.0 , vNdotL) * diffuseLight) ;
}
```

Listing 3: Simple Lambert Fragment Shader Automatically Created

```
varying vec3 lambert ;
void main()
{
        gl_FragColor = vec4(lambert, 1.0);
}
```

3.4.2 Search performance

The implementation has been run with 1 to 5 different variants for 8 parts (totalling 8 to 8^5 solution parts). The tests have been made on a computer with 2.00 GHz Intel P4-processor and 1 GB RAM, and show that the system can get a solution within $\frac{1}{50}$ of a second if all parts have two different variants. With three variants the search will give a small but noticeable decrease in rendering speed (see table 1). It should be noted that it is not very likely that many parts have different variants. Nor is it likely that searching will be done inside a critical render loop.

Parts	Variants of each part	Run time in seconds
8	1	0.156
8	2	2.109
8	3	11.907
8	4	43.719
8	5	128.719

Table 1: Search time for 100 searches with varying number of part variants

3.4.3 Maya extension

The implementation has also been added as an extension to Alias Systems Maya 6.0 [Folkegard 30/09/2004], allowing a user to work directly with hardware shading in the Maya work environment. The extension connects the Shader Combiner system to the Maya user interface. Solution parts can be created, viewed and selected to generate the desired effects on polygon objects. The result of each generated shader is immediately visible in the work environment, and the final code can be read and exported from within Maya. All global variables of each shader can be connected to the rest of the Maya system, including the expression and scripting engine and the animation functions. The main use for this extension is as a tool for graphics algorithm artists developing real-time effects for games and research purposes, but it can also be used to enhance hardware rendered effects for film and TV.

4 Discussion

The algorithm is sufficient for the purpose of finding working program pairs for real time shaders. By focusing only on solution parts useful at a given moment the algorithm works faster than it would have done if it had gone through all existing parts. The algorithm can improve by rejecting unsuitable solutions before all solutions are found, but finding such an optimized search algorithm is outside the domain of this work.

The system is well suited to quickly create shaders. For easy control of the result it has been useful to create a node representing the preconditions and final operations of the result, and then to require that part in all solutions. Since the system generates directed acyclic graphs, the user is urged to design shaders in the same fashion. That way the design is easier to grasp, and circular dependencies are avoided.

4.1 Future development

For substantial usefulness in large scale development of real time shaders, the system needs to be independent of language. The search algorithm should be modified so it rejects unsuitable solutions in earlier steps of the search, thereby avoiding the problem of huge time costs when encountering many alternative search paths. Shaders being too large to run in hardware could also be divided by the system, much as the method suggested by Chan et. al [Chan et al. 2002].

Currently all solution parts consist of static functions. If output from one shader shall be used as input for another, the receiving shader must state this. In order to be fully flexible the system should be able to generate compound shader trees where the connection between nodes and their internal order can be specified arbitrarily. This can be made possible if the solution parts are allowed to receive and give out arbitrary parameters, which would mean that the system creates its own static functions from a description of a general function. If this succeeds there is a possibility to use the system to assemble other data, for example regular procedural code in systems for visual programming.

5 Conclusion

Shader programming can be made less redundant by dividing programs into parts. Thereby hardware shading is made more modular and more flexible. By using the shader combiner system presented here, an application can use a richer set of real time shaders without having to put a lot of time into developing these. Dividing shader programs in smaller parts can make the different concepts of shading very clear. Work can thereby be focused on using the concepts for one's own artistic purposes and for quickly trying out new shading ideas. The implementation has a sufficient speed for efficient use in real-time applications, making it useful for both shader development purposes and as a subsystem in games and visualizations.

References

BLEIWEISS, A., AND PREETHAM, A., 2003. Ashli – Advanced Shading Language Interface. http://www.ati.com/developer/eurographics2003/AshliViewerNotes.pdf.

CHAN, E., NG, R., SEN, PRADEEP, S., PROUDFOOT, K., AND HANRAHAN, P. 2002. Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware. In *Graphics Hardware(2002)*, 69 – 78.

FERNANDO, R., AND KILGARD, M. J. 2003. *The Cg Tutorial*. Addison–Wesley.

FOLKEGARD, N., 03/06/2004. Shadercombiner implementation. http://sourceforge.net/projects/shadercombiner/.

FOLKEGARD, N., 30/09/2004. Hardware Shader Combiner for Maya. available through Creative Media Lab, Gävle.

HAST, A. 2004. *Improved Algorithms for Fast Shading and Lighting*. PhD thesis, Uppsala Universitet.

KESSENICH, J., BALDWIN, D., AND ROST, R. The OpenGL Shading Language.

LIGHTWORKS, 20/05/2004. Programmable Hardware Shading In Applications – Challenges and Solutions. http://www.lightworkdesign.com/news/media/ProgrammableShadingInApplications.pdf.

MARK, W. R., GLANVILLE, S. R., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. In *ACM Transactions on Graphics*, ACM, 896 – 907.

MCCOOL, M. D., DU TOIT, S., POPA, T., CHAN, B., AND MOULE, K. 2004. Shader Algebra. In *ACM Transactions on Graphics*, ACM, 787 – 795.

ROST, R. J. 2004. *OpenGL Shading Language*. Addison–Wesley.

RTZEN INC. RT/shader.

PyFX – An active effect framework

Calle Lejdfors and Lennart Ohlsson
Dept. of Computer Science
Lund University, Sweden
{calle.lejdfors,lennart.ohlsson}@cs.lth.se

Abstract

The programmability of modern graphics processing units (GPUs) provide great flexibility for creating a wide range of advanced effects for interactive graphics. Developing such effects requires writing not only shader code to be executed by the GPU but also supporting code in the application where the effect is to be used. This support code creates dependencies between effects and the applications that use them, making it harder to evolve applications and to reuse effects. Existing effect frameworks, such as DirectX Effects and CgFX, can only provide partial encapsulation because they consider effects as passive data structures. In this paper we present an effect framework written in an ordinary scripting language where effects are active entities. This makes it possible to completely encapsulate both shaders and support code thereby minimizing the dependencies to the application.

1 Introduction

The availability of programmable graphics processors has made procedural effects a key ingredient in real-time graphics productions. Where content creation previously was mainly the combination of a wide range of different kinds of artwork such as geometric models, textures, and motion data, it now also has to include algorithmic development. Writing the shader code to be executed on the graphics processors is something which traditionally is not part of an artist's skill set. Instead this new development model requires a closer relationship between artists and shader programmers. Previously programmers of interactive graphics applications were primarily concentrated with loading and displaying content created by the artists in an efficient and correct manner, a task which is handled fairly independent of the actual content. But with programmable graphics processors the roles of artists and programmers become more intertwined. When the artist conceives of a visual effect it is the programmers job to supply shader programs and the necessary modifications to the application for achieving that effect. But once written, the shader program typically requires actual textures and parameter values and it is the artists job to supply that.

For efficient collaboration it is important, to both artists and programmers, that the graphical effect is a well-defined entity. It should include all relevant resources and functionality, both shader code and application support, required for correct operation. This need for encapsulation is the motivation behind technologies such as the DirectX Effects by Microsoft and CgFX by NVIDIA. In these frameworks the notion of an *effect* is used as the key unit of abstraction. But although these technologies provide a number of features which improve the handling of effects they still require a substantial amount of application support. All but the most trivial effects have dependencies in the application that use them.

The effect framework presented in this paper aims to provide *complete encapsulation* of effects in the sense that specific support code avoided and parameter passing is made with the most unobtrusive mechanism possible. We have implemented a prototype which uses Python both for the implementation of the framework and to express the effects themselves. This paper is focused on the implementation of the framework and its application interface, whereas the benefits of writing effects in Python is described in more detail elsewhere [Lejdfors and Ohlsson 2004].

1.1 Related work

The focus on complete effects is different from most other approaches. Most real-time shading language research has been focused on mapping high-level shading languages to real-time shading hardware. Research was initiated by Cook [Cook 1984] with the introduction of shade trees, which spawned a number of shading languages such as RenderMan [Hanrahan and Lawson 1990] or Perlins image synthesizer [Perlin 1985]. These languages were originally used for off-line shading but with advances in hardware Peercy et al showed that it was possible to execute RenderMan shaders on an extended OpenGL 1.2 platform by viewing the graphics hardware as a SIMD pixel processor [Peercy et al. 2000]. Olano et al presented an alternative approach with the pfman language [Olano and Lastra 1998] for the Pixelflow rendering system [Molnar et al. 1992], a flexible platform based on image composition which, unfortunately, bears little resemblance to the GPUs of today.

The computational model of separating per-vertex and per-pixel computations was introduced by Proudfoot et al [Proudfoot et al. 2001] which allowed the to efficiently map shader programs to hardware. This separation is used explicitly the in real-time shading languages used in the industry today: Cg by NVIDIA [Mark et al. 2003], HLSL by Microsoft [Gray 2003] and OpenGL shading language [3D 2002] introduced with OpenGL 2.0. This is also the case with the Sh language [McCool et al. 2002][McCool et al. 2004]; a shading language embedded in C++ which provides a number of powerful high-level features for shader construction. Another embedded shading language is Vertigo [Elliott 2004] which uses the purely functional language Haskell as a host language to provide a clean model for writing shaders for generative geometry.

All these efforts have focused on various aspects of shader programming but the writing of effects containing multiple shaders have not received the same amount of attention. The Quake3 shader model [Jaquays and Hook 1999] provides a rudimentary interface for controlling the application of multiple textures. DirectX Effects [Dir] extend the HLSL shading language and introduce a richer, more powerful interface for controlling the rendering pipeline. NVIDIA provide a superset of this functionality with their CgFX framework [CgF], based on Cg. This is further elaborated on in Section 1.3.

1.2 Shader programming

To demonstrate the issues involved in the implementation of shader based effects and how an active framework like PyFX can alleviate these problems we will use a running example throughout this

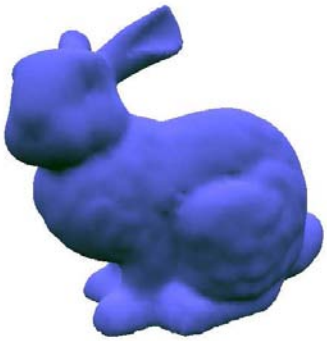


Figure 1: Hemispheric lighting on bunny

paper. The description of this example will be fairly detailed because the causes of application dependencies and need for support can often be found in those details which would usually be omitted in a more concise description. The example we use is the lighting model known as hemispheric lighting, where the idea is to give a contribution of indirect light as a mixture of light from the sky and light from the ground. A given point is colored depending on orientation of its surface normal, the more it is oriented towards the sky the more light from the above light source it receives, and vice versa. The effect of using this model can be seen on the bunny in Figure 1. A shader program which implements this model can be written in Cg as

```
void main(float4 position : POSITION,
          float4 normal   : NORMAL,

      out float4 clipPosition : POSITION,
      out float3 color        : COLOR,

  uniform float4x4 ModelViewProj,
  uniform float4x4 ModelViewIT,
  uniform float4x4 WorldView,
  uniform float3   MaterialColor,
  uniform float3   SkyColor,
  uniform float3   GroundColor)
{
  clipPosition = mul(ModelViewProj, position);
  float4x4 ModelWorldIT = mul(WorldView,ModelViewIT);
  float3 worldNormal = mul(ModelWorldIT,normal).xyz;

  worldNormal = normalize(worldNormal);
  color = lerp(GroundColor, SkyColor,
               (worldNormal.y + 1)/2)*MaterialColor;
}
```

Listing 1: Hemispheric lighting in Cg

This shader program is a vertex shader. It computes the vertex normal in world-space `worldNormal` by using the inverse transpose of the model-world transform `ModelWorldIT`. The amount of incident light of the vertex is then computed by linear interpolation lerp of the sky and ground color where the weighting factor is determined the y-component world-space normal. The incident light is weighted by the material properties of the object. Finally, as required by all vertex programs the clip-space coordinates `clipPosition` are computed using the projection matrix `ModelViewProj`.

The parameters to the program which are marked as `uniform` are those which are constant for the duration of the shader program, whereas the other parameters vary over the vertices of the mesh. The extra field (`POSITION`, `NORMAL`, and `COLOR` in this example), known as the *semantic* of the parameter specify how they are mapped to application data. For example, an `in` parameter with semantic `NORMAL` is specified using OpenGL's `glNormal*` calls from the application.

Shaders require application programmers to write support code for every shader to be used. In order to access shader program parameters an application level identifier is needed. Accessing the parameter identifiers of our example shader from C++ would be as follows.

```
cg_mvp = cgGetNamedParameter(cg_prog,"ModelViewProj");
cg_mvit = cgGetNamedParameter(cg_prog,"ModelViewIT");
cg_wv = cgGetNamedParameter(cg_prog,"WorldView");
cg_materialColor = cgGetNamedParameter(cg_prog,
                                       "MaterialColor");
cg_groundColor = cgGetNamedParameter(cg_prog,
                                     "GroundColor");
cg_skyColor = cgGetNamedParameter(cg_prog,"SkyColor");
```

Listing 2: Finding parameter identifiers

Each time the shader is used it must be bound after which each parameter has to be set to its current value using the corresponding Cg parameter identifier. The target for which the shader program has been compiled, called the profile of the program, must also be enabled.

```
cgGLBindProgram(cg_prog);

cgGLSetStateMatrixParameter(cg_mvp,
    CG_GL_MODELVIEW_MATRIX,CG_GL_MATRIX_IDENTITY);
cgGLSetStateMatrixParameter(cg_mvit,
    CG_GL_MODELVIEW_MATRIX,
    CG_GL_MATRIX_INVERSE_TRANSPOSE);
cgGLSetMatrixParameterfr(cg_wv,
    camera->inverseTransform());
cgGLSetParameter3fv(cg_materialColor, MaterialColor);
cgGLSetParameter3fv(cg_groundColor, GroundColor);
cgGLSetParameter3fv(cg_skyColor, SkyColor);

cgGLEnableProfile(cg_profile);
```

Listing 3: Binding shader program and setting parameters

Changing the parameters of the effect at run-time amounts to changing the local variables used here, for example `MaterialColor`, `SkyColor`, and `GroundColor`.

This code can be compiled and delivered together with the shader code as a complete package which can be used by the artist. However, there are limitations with this approach. All but the most trivial shaders require support code for setting parameters and renderer pipeline states. This support code is specific to each application due to differences in how textures are loaded and accessed, renderer pipeline state are set, *etc*. This gives unwanted dependencies between shaders and applications. Encapsulating these dependencies is a difficult problem since different applications have very different notions of what is important, for instance an artist's tool must be able to provide GUI components for manipulating the shader whereas an engine is primarily concerned with efficiency.

This encapsulation is made even more difficult when using shaders written by an external party. Externally written shaders use different interfaces but must still be accessible in the same manner as in-house developed ones in order to provide a unified working model for both artists and developers. The amount of work needed to adapt such shaders can often be too large.

1.3 Effects

The problems associated with using shaders as shown above are caused by a lack of encapsulation. Information associated with the shader and necessary for the shader to work is mixed with application code and not packaged together with the shader itself. This has called for a new level of abstraction and a new kind of entity to do the encapsulation. These entities are known as *effects*.

Today there are two major effect frameworks in use, the DirectX Effects by Microsoft [Dir] and CgFX by NVIDIA [CgF]. Both provide a text-based format where shader code, parameters and pass specifications are written in one file. This file is loaded by the application and compiled for the current run-time platform. The two formats are very similar and can in many instances be used interchangeably. Using CgFX the hemispheric lighting example can be implemented as:

```
float3 MaterialColor = { 1.0, 1.0, 1.0 };
float3 SkyColor = { 0.5, 0.5, 1.0 };
float3 GroundColor = { 0.0, 0.1, 0.0 };

float4x4 ModelViewProj : MODELVIEWPROJ;
float4x4 ModelViewIT   : MODELVIEWIT;
float4x4 WorldView     : WORLDVIEW;

shader code as in listing 1

technique Hemispheric {
  pass p0 {
    VertexShader = compile vs_1_1 main(ModelViewProj,
                                       ModelViewIT,
                                       WorldView,
                                       SkyColor,
                                       GroundColor);
  }
}
```

Listing 4: Hemispheric lighting in CgFX

This effect declares three parameters which are intended to be set at design time, `MaterialColor`, `SkyColor`, and `GroundColor`, and three parameters that are intended to be set at run-time by the application: `ModelViewProj`, `ModelViewIT`, and `WorldView`. The design-time parameters, also known as *tweakables*, may have associated annotations which can be used by design tools to automatically provide a suitable user interface for setting the parameter. For example a color picker control may be used to set the value of a color parameter. Run-time parameters on the other may have *semantic* identifiers associated with them, and similar to shader semantics, their purpose is to specify the mapping to application data without relying on parameter name. Instead an application can define a number of semantic identifiers which may be used in the effect.

Following the declaration of the effect parameters is the shader code. It is identical to Listing 1 and is therefore omitted here. Finally the effect declares a so called *technique* which describes number of rendering passes needed and the render states to be used in each pass. In this case there is a single rendering pass and in that pass the vertex shader `main` is to be compiled for the shader profile `vs_1_1` and the uniform shader parameters should have the values of the corresponding effect parameters.

Once loaded an effect can be used in the application like this

```
unsigned int numPasses;
effect->Begin(&numPasses, 0);
for (unsigned int p = 0; p < numPasses; p++)
{
  effect->Pass(p);
  renderMesh(mesh);
}
effect->End();
```

Listing 5: Effect usage with CgFX

The textures used by an effect are generally declared as tweakables where an annotation is used to specify the filename.

```
texture colorTexture : DiffuseMap <
   string File = "default_color.dds";
>;
```

Using a texture in a shader program is done indirectly through something called a *sampler* which specifies how the texture is accessed. Declaring a simple 2-dimensional sampler using linear minification and magnification filters for the above texture we write

```
sampler2D colorSampler = sampler_state {
  Texture = <colorTexture>;
  MinFilter = Linear;
  MagFilter = Linear;
};
```

This sampler is then passed to a shader program just as any other parameter.

Effects provide a number of mechanisms for separating applications from shaders. First, the effect format give a clear, high-level, and concise specification of shader programs, textures, and render states. This includes a unified method for handling multipass effects as well as having multiple implementations (fixed-function fall backs *etc.*) of the same visual effect. This specification is independent of the target architecture on which the effect is to run.

Second, tweakables provide the artist with a method for setting parameters at design time. This reduces support code since the engine only needs to concern itself with providing run-time parameters such as projection matrices *etc*.

Third, user-defined semantics provides a method for the engine to provide such run-time parameters. The application defines a number of semantic identifiers which it support and this creates an rudimentary interface to effects which, together with default values for parameters, relieves the effect developers of writing per-effect support code (cf. listings 2 and 3).

However, as in the case with using shaders directly, there still exist a problem of encapsulation. The application defines an interface for the effects by using user-defined semantics. This interface is fixed, and this limits the number of shaders that may be expressed and used within a single application.

2 PyFX

The limitations in encapsulation of existing effect frameworks is due to the fact that effects are passive entities, text files, which are operated on by the application, which is the active party. If this relationship could be reversed so that effects are active instead, a better interface can be built where they can be responsible for retrieving the data they need from applications rather than the other way around. To achieve this reversed flow of control the effects must be embedded in a context which can do actual execution on their behalf.

2.1 PyFX overview

We have used Python, an existing scripting language to develop an active effect framework called PyFX. The current implementation supports applications using OpenGL and shaders written in Cg and its feature set closely resembles that of CgFX. In PyFX however, Python is used both to implement the framework and to write the effects themselves.

In an object-oriented language, it is natural to represent different effects as subclasses to a common effect base class. The subclasses implement specific functionality whereas functionality common to all effects are inherited from the base class. The object-oriented model also provides a natural mapping to the collaborative workflow between programmers and artists. Effect programmers write new effects by making new effect subclasses, whereas the artist provides textures, sets parameters, etc. to make effect instances from existing classes.

Below is the hemispheric lighting example written in PyFX. It shows the Python class `Hemispheric` as a subclass of the general `Effect` class.

```
class Hemispheric(Effect):
   vs = Cg("""
      shader code as in listing 1
   """)

   SkyColor    = (0.5, 0.5, 1.0)
   GroundColor = (0.0, 0.3, 0.0)

   def __init__(self,
                MaterialColor = (1.0, 1.0, 1.0)):
       Effect.__init__(self)

       self.MaterialColor = MaterialColor

       self.technique = [Pass(VertexShader = vs())]
```

Listing 6: Hemispheric lighting in PyFX

The declaration has two main parts: the class variables and the constructor (the `__init__` member). The first class variable `vs` contains the shader program as a string wrapped by an instance of a Python class called `Cg`. and the other two class variables `SkyColor` and `GroundColor` are simply effect parameters. The class constructor, which creates new instances of the class, takes one additional effect parameter `MaterialColor` as an argument. The ability to differentiate between class variables and instance variables allows the effect writer to indicate that some parameters are intended to be the same for all instances of the class whereas other parameters may be different. The constructor body calls the superclass constructor and sets the instance variable `MaterialColor` of the object. Finally, the instance variable `technique` is set to specify that this is a single pass effect and that the pass should use the shader `vs` as its vertex shader.

Having instantiated this effect, for example like this

```
effect = Hemispheric(MaterialColor = (0.0,0.0,1.0))
```

it can be applied to a mesh by

```
while effect.hasMorePasses(mesh):
    renderMesh(mesh)
```

The `Effect` member function `hasMorePasses` does setup for each pass of the effect and also specifies how many times the mesh needs to be rendered.

Having applied effects to meshes the next issue is the passing of information from the application to the effect and its shader. In PyFX this data can be passed through a number of different channels.

The most obvious way is through constructor parameters when the Hemispheric effect is instantiated. The example above shows how `MaterialColor` is set to the color blue.

In the hemispherical lighting example the constructor parameters correspond exactly to an instance variable of the effect. Another method of passing data to the shader is to assign new values to this variable. For example

```
effect.MaterialColor = (0.5, 0.5, 1.0)
```

changes material color so that it is now light blue. Similarly class variables can also be assigned new values

```
Hemispheric.GroundColor = (0, 0, 0)
```

The framework then make sure that these changes are made available to the shader code.

Another type of parameters are the transformation matrices used by the effect; `ModelViewProj`, `ModelViewIT` and `ViewWorld`. The matrices `ModelViewProj` and `ModelViewIT` can be retrieved from the OpenGL rendering pipeline and in PyFX this is handled automatically.

The third parameter `ViewWorld` needs special treatment. It is the inverse camera transform, used by the effect to compute the `ModelWorld` transform, neither of which can be automatically retrieved from the pipeline. It must therefore be provided by the application. Since this parameter is the same for different instances it makes sense to make it a class variable. However, it is even more general than that since you could easily think of other effects that might need it. In this case we can therefore set it as a class variable on the `Effect` base class, for example:

```
Effect.ViewWorld = camera.inverseTransform()
```

Yet another method for passing data from the application is when the effect needs additional data at each vertex, *i.e.* non-standard varying parameters. In our hemispherical lighting example this is not the case, but a more advanced version of hemispheric lighting can used to illustrate this case [Hem]. This version use additional per-vertex mesh data, called the *occlusion factor*, which determine the amount of hemispheric light which reach the point in question. If the shader program has the following prototype

```
void main(..., float OcclusionFactor : COLOR )
```

Then, if the mesh has an array member `OcclusionFactor`, PyFX will automatically bind this to the varying parameter with the same name.

2.2 PyFX details

2.2.1 Techniques

The structure of PyFX effects is inspired by that of DirectX Effects and CgFX frameworks. As in these each effect contain one or more techniques. Each technique contain a number of passes which are to be run consecutively. Each render pass has associated render states specifying the necessary pipeline states required to run the pass. Specifying that back-face culling should be disabled while alpha-blending is enabled is written in CgFX as

```
pass p0 {
  CullMode = NONE,
  AlphaBlendEnable = True
}
```

In PyFX the same render pass specification would look like

```
Render(CullMode = None,
       AlphaBlendEnable = True)
```

A single technique effect for CgFX is shown in listing 4. The corresponding effect in PyFX is given in listing 6. Providing two techniques `Hemispheric` and `Ambient` in CgFX is done by providing multiple technique blocks

```
technique Hemispheric {
  pass p0 {
    VertexShader = compile vs_1_1 main();
  }
}

technique Ambient {
  pass p0 {
    Color = <ambientColor>;
  }
}
```

The same would be written in PyFX as

```
technique = {}
technique['Hemispheric'] = [
    Render(VertexShader = vs())]
technique['Ambient'] = [
    Render(Ambient = AmbientColor)]
```

2.2.2 Textures

Texturing in PyFX is, as in CgFX, divided into textures and samplers. Declaring the same texture and sampler as above (Section 1.3) in PyFX would be written as

```
colorTexture = Texture(filename="default_color.dds")
colorSampler = Sampler(colorTexture,
                       MinFilter = Linear,
                       MagFilter = Linear)
```

This sampler can then be used either by a shader program, using parameter resolution, or in the fixed-function pipeline by

```
Render(Texture0 = colorSampler)
```

Multi-texturing is naturally supported and when using multiple textures in a shader program this is automatically handled by the shader parameter resolution code. For fixed-function effects the different texturing-units are accessible via

```
Render(Texture0 = colorSampler,
       Texture1 = lightMapSampler)
```

where `colorSampler` and `lightMapSampler` are two samplers with appropriate settings.

2.2.3 Shaders

Shaders are provided via strings wrapped with classes providing information on the type of shader code contained in the string. Sometimes it is useful to specify the target for which a given shader should be compiled. This can be achieved via

```
Render(VertexShader = vs(target=arbvp1))
```

Also, passing explicit parameters to shader programs can be done by adding keyword arguments to the shader invocation. Suppose we have an outlining effect which draws a gradually more transparent outline around an object. This effect should run multiple passes with the same shader program (called `outline`) but with a parameter `offset` determining the size and opacity of the outline

```
[Render(VertexShader = outline(offset=1.0)),
 Render(VertexShader = outline(offset=0.75)),
 Render(VertexShader = outline(offset=0.5)),
 Render(VertexShader = outline(offset=0.25))]
```

Listing 7: Setting compile-time parameters

The same thing can be expressed in CgFX but the result is more verbose since every shader parameter must be passed explicitly.

If a shader program source code `shader` contains multiple programs, say a vertex shader `shadeVertex` and a pixel shader `shadePixel`, these programs entries can be accessed by the corresponding methods on the shader object

```
Render(VertexShader = shader.shadeVertex(),
       PixelShader = shader.shadePixel())
```

2.2.4 Parameter resolution in PyFX

Application level variables having the same name as the shader program parameters are used as arguments to the shader program. These arguments are defined in one of the following places:

- Either it is a compile-time parameter to the shader program (see listing 7), or
- an attribute of the effect object, or
- an attribute on the mesh currently being rendered, or, lastly,
- a member of a predefined set of state parameters giving access to current pipeline states.

Attributes of the effect instance include both instance parameters, such as the material color parameters above (Section 2.1), and class parameters, `SkyColor` and `GroundColor` above. As usual the class scope includes the scope of its superclass making the `WorldView` transform accessible to the shader programs. In the above examples the `OcclusionFactor` is a mesh attribute and them `ModelViewProj` and `ModelViewIT` matrices are both pipeline state parameters.

If there are multiple variables with the same name the order of precedence is that compile-time parameters take precedence over instance attributes, which take precedence over class variables. Effect class variables take precedence over mesh attributes and state parameters are used last.

This gives a natural correspondence between parameters to the shader program and application data. Setting effect class-specific values amounts to setting effect class-variables whereas effect instance-specific values are set by setting the appropriate attribute on the effect instance in question. Effects take a more active role since they are allowed to extract information from the mesh currently being rendered thus minimizing the amount of application level dependencies.

The mapping is recursive so the following Cg shader program

```
struct Light {
  float3 position;
  float4 color;
};

void main(..., uniform Light light) { ... }
```

will use `position` and `color` member of the application level variable `light`.

2.2.5 Name maps

The lookup scheme above gives great flexibility in both writing and using effects. However when dealing, for instance, with third-party effects a name-based lookup is not always sufficient since naming conventions may differ. Suppose we wish to use an effect which uses the name `DiffuseMap` where our application use `DiffuseTex`. An obviously unattractive solution would be to add `DiffuseMap` to our code and make sure to update it each time we change `DiffuseTex`.

PyFX solves this problem by having user defined *name maps*. The `Effect` class allows us to pass a dictionary of how parameter names at the shader level should be mapped to parameter names at the application level. Defining a dictionary containing our mappings and passing it to the effect nicely handles this.

```
myNameMap = {'DiffuseMap' : 'DiffuseTex'}

effect = SomeTexture(nameMap = myNameMap)
```

A request for the `DiffuseMap` will now be automatically translated to a requests for `DiffuseTex`.

2.2.6 Language embedding

The fact that Python is used to write effects and not only for implementing the framework is convenient but not strictly necessary. It would have been possible to write an interpreter and for example use the CgFX format. However, the complete *embedded* of effects in Python has the advantage that all the ordinary language features such as lists, tuples, loops, functions, dictionaries, list comprehension, etc. are available to the effect writer[Lejdfors and Ohlsson 2004]. As a very simple example we could have used a list comprehension to write the pass specification of the outline effect (listing 7) as

```
[Render(VertexShader=outline(factor=f))
                        for f in [1.0, 0.75, 0.5, 0.25]]
```

2.2.7 Module-style effects

When using concrete subclasses of `Effect` the application needs to know about every such class at compile-time, something known as the *library problem*. This is clearly not desirable in a graphics application and it was one of the problems effects where created to alleviate. In traditional object-oriented design it is solved by introducing an abstract factory for handling instantiation of concrete subclasses [Gamma et al. 1994]. However, using the flexibility of Python we can provide a method which simultaneous solves this problem while giving a cleaner and more direct syntax for declaring effects. Effects can be implemented simply as Python modules which can be loaded by

```
effect = Effect('Hemispheric')
```

This loads the `Hemispheric` effect module which can be used just as any other effect. Note however, that since the actual subclass is not known setting class variables such as `GroundColor` (cf. Section 2.1) is not possible.

2.2.8 Image processing

PyFX also provides a mechanism for specifying render targets other than the frame buffer to which rasterization should occur. Furthermore it is possible to have passes which do not render geometry but instead do shader based image processing. These two features, which are not available in CgFX or DirectX, allow effects such as blurring, edge detection, image compositing *etc*. to be expressed in an application independent manner.

3 Implementation

PyFX is implemented on top of PyOpenGL [PyO] and a SWIG [SWI] generated interface to the Cg runtime library. The implementation consists of about 800 lines of Python code. The bulk of it is concerned with basic functionality needed in any effect framework such as loading and binding textures, compiling shader programs, and initializing OpenGL extensions. The remaining part implements the distinguishing features of PyFX, *i.e.* mapping declarative state specification to function invocations and performing parameter resolution. This part is remarkably small, only about 10% or 80 lines of code. This compactness is possible because of Python's dynamic object model and introspection facilities.

3.1 Renderer management

The entry point of the PyFX framework is provided by the top-level `Effect` class. It is essentially a container for other objects, *i.e.* techniques, passes, textures, samplers, and shaders. These classes interact with the underlying graphics API through a global `RenderState` singleton class which implements manipulation of the renderer pipeline state. The majority of its methods correspond one-to-one to the available state variables. For instance the `CullMode` state is implemented as

```
class RenderState:
  ...
  def CullMode(self, val):
      if val:
          glEnable(GL_CULL_FACE)
          glFrontFace(val)
      else:
          glDisable(GL_CULL_FACE)
```

When a `Render` is activated it instructs the `RenderState` object `state` to change the state of the rendering pipeline. This is done by mapping every state specified in the pass object to a method invocation. For example, a pass specified by

```
Render(Color = (1.0, 0.0, 0.0),
       CullMode = None)
```

will result in the following method calls on:

```
state.Color((1.0, 0.0, 0.0))
state.CullMode(None)
```

Doing this mapping is the responsibility of the `Render` class and by using the dynamic introspective features in Python, it can have a very small implementation:

```
class Render:
  def __init__(self, **kwords):
      self.kwords = kwords

  def use(self, state):
      for s,v in self.kwords.items():
          marshalFX(state,s,v)
```

The `marshalFX` maps the state name `s` to the proper method name and calls this method with argument `v`. It is similar to the marshaling used by RPC (remote procedure calls), whereby serialized data (dictionary tuples) are converted to method invocations. Implementing `marshalFX` is a two-liner:

```
def marshalFX(obj, name, *args):
    method = getattr(obj,name)
    return method(*args)
```

3.2 Texture and sampler state

The class `Texture` provides an encapsulation similar to `RenderState` but for the available texture states such as filtering, texture coordinate wrapping, etc. The texture state information is maintained by the corresponding `Sampler` and it is responsible for marshaling this information to method invocations on the `Texture` object.

When a `Sampler` is used by either the fixed-function pipeline or by a shader the framework allocates a free texture unit and asks the sampler to bind itself to that unit.

3.3 Shaders

Just as samplers are responsible for performing binding textures and setting texture states, every `Shader` object is responsible for performing its own loading, binding, compilation, and parameter resolution. This implementation is actually contained in subclasses for different shader programming languages. Currently the only subclass implemented is `Cg`.

When the pass specifies a vertex or fragment shader the `state` object instructs the shader to bind itself. A shader binding itself includes setting the value of every parameter needed by the shader. The mapping scheme of PyFX between parameters and application variables is implemented by a `Resolver` object whose responsibility it is to search the effect and mesh name spaces as well as providing name mapping (Section 2.2.5). The resolver searches a list of objects for a given attribute, optionally transform the attribute name via the name mapping dictionary:

```
class Resolver:
   def __init__(self,nameMap,*objs):
       self.nameMap = nameMap
       self.objs = objs

   def __getattr__(self,attr):
       if self.nameMap.has_key(attr):
          attr = self.nameMap[attr]

       for obj in self.objs:
          if hasattr(obj, attr):
              return getattr(obj, attr)
```

The `Cg` class use the resolver to locate shader parameters and set these by invoking the corresponding CgGL functions. For simple variables the `cgGLSetParameter`-family of functions are used. Aggregate parameters, such as arrays and structs, are handled by iterating over the members and setting each element recursively.

4 Conclusions and future work

The most prominent features provided by the PyFX framework is the decoupling of effects from the application. This "activation" of an effect, enabling it to obtain needed data from *e.g.* the current mesh without the need to introduce application level support code, greatly reduces dependencies between effects and the application. Using this activation together with the introspection features of Python gives a natural mirroring between data at the application level and data at the level of shader programs. This also eliminates the need for user-defined semantics since there is no longer any need to provide *ad hoc* hooks for applications to provide specialized data and operations. Instead the object-oriented extensible nature of the host programming languages can be used to provide this functionality natively at the effect level.

There are some limitations however, in the current implementation of PyFX. Support for manipulating fixed-function effect parameters is limited. Consider a simple effect such as

```
class SimpleColor:
    color = (1,0,0)
    technique = [Render(Color=color)]
```

Manipulating the `color` attribute of this effect will not have the desired effect, the color used for drawing will remain red. The reason why the parameter resolution algorithm (Section 2.2.4) can not be applied in this case is that it requires access to the parameter names. These names are only available to shader based effects where they are supplied by the Cg run-time library.

The overall purpose of PyFX is to be a flexible tool for investigating what kind of features and functions are needed to make effect programming as easy and productive as possible. Future work includes investigating how effects can be combined efficiently at run-time allowing, for instance, stencil-buffer shadow algorithms to coexist with other visual effects.

Acknowledgement

The authors wish to thank Benny Åkesson and Andreas Hansson for help with implementing the framework as well as providing a number of interesting effects testing various aspects of PyFX. We also extend our thank to Chuck Mason for providing the initial Python interface to Cg.

References

3D LABS. 2002. *OpenGL 2.0 Shading Language White Paper*, 1.2 ed., February.

CgFX 1.2 Overview. `http://developer.nvidia.com/`.

COOK, R. L. 1984. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, 223–231.

DirectX SDK Documentation. `http://msdn.microsoft.com/`.

ELLIOTT, C. 2004. Programming graphics processors functionally. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, ACM Press, 45–56.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns*. Addison-Wesley.

GRAY, K. 2003. *DirectX 9 programmable graphics pipeline*. Microsoft Press.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, ACM Press, 289–298.

Hemispheric lighting. Example in DirectX 9 SDK documentation. MSDN Library, `http://msdn.microsoft.com`.

JAQUAYS, P., AND HOOK, B. 1999. *Quake III Arena Shader Manual*, 12 ed. Id Software Inc., December.

LEJDFORS, C., AND OHLSSON, L. 2004. Tools for real-time effect programming. Submitted to publication.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph. 22*, 3, 896–907.

MCCOOL, M., QIN, Z., AND POPA, T. 2002. Shader metaprogramming. In *Graphics Hardware*, T. Ertl, W. Heidrich, and M. Doggett, Eds., 1–12.

MCCOOL, M., TOIT, S. D., POPA, T. S., CHAN, B., AND MOULE, K. 2004. Shader algebra. In *To appear at SIGGRAPH 2004*, 9 pages.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. Pixelflow: high-speed rendering using image composition. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, ACM Press, 231–240.

OLANO, M., AND LASTRA, A. 1998. A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, 159–168.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 425–432.

PERLIN, K. 1985. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM Press, 287–296.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 159–170.

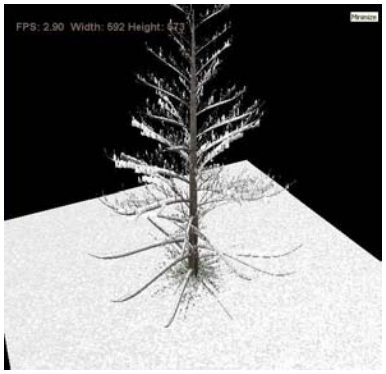
PyOpenGL project. `http://pyopengl.sf.net/`.

SWIG project. `http://www.swig.org/`.

Real-time Rendering of Accumulated Snow

Per Ohlsson
Uppsala University

Stefan Seipel
Uppsala University



Abstract

This paper presents a method of computing snow accumulation as a per pixel effect while rendering the scene. The method is similar to the shadow mapping method for shadow calculations. A depth buffer is used to find out how much snow a particular surface should receive. The amount of snow is then modified depending on the slope of the surface. To render the snow in a convincing way 3D noise is utilized for the lighting of the snow surface.

Keywords: snow, computer graphics

1. Introduction

Snow is a common phenomenon in nature. It has the ability to completely transform the mood of a scene, turning a rocky landscape into a gentle sear of white tranquillity. But even though snow is a common occurrence in many places of the world, snow accumulation in real-time seems to be an area that has still to be explored. Few attempts have been made, and the methods demand lots of work do be done on the scene modelling before they can be used.
Usually when snow occurs in interactive simulations it is either modelled by an artist, or just plain textures drawn upon the original scene.
The ability to automate the snow accumulation process in real-time would give us the ability to add snow to any scene, and by doing this produce beautiful settings for any kind of interactive environment. This would enable artists to just create non snow scenes and then let the algorithm create a snow cover, effectively creating two scenes. It would greatly decrease the amount of work needed to model a snow covered scene.

2. Related Work

The subject of offline snow rendering has been looked into quiet throughout.
Paul Fearing [2000] presented a method to create beautiful snow scenes in 'Computer Modelling Of Fallen Snow'. Unfortunately each frame took very long time to render making the method not viable for adaptation to any kind of real-time situation. It worked by first tracing snow paths from the ground upwards towards the sky, and by that accumulating the snow. In a second stage the stability of the snow was calculated, moving snow from unstable areas to stable,
In the paper "Animating Sand, Mud and Snow" Summer et al. [1998] describe a method for handling deformations of surfaces due to external pressures. In their algorithm the surface is divided into voxels containing different height values indicating the height of the current surface. This grid is used to calculate the deformation of the surface when interacting with objects.
Nishita et al. [1997] use metabals and volume rendering for their snow accumulation and rendering, taking into account the lights path and scattering through the snow. All this makes it less suitable for real-time modification.

Snow rendering in real-time has received less attention.
Haglund et al. [2002] propose a method where each surface is covered with a matrix containing the snow height at that position. The snow accumulation is handled by dropping snow flakes, represented by particles, from the sky. In the place where a snow flake hits the ground the height value gets increased. Triangulations describing the snow are then created from the matrices containing the height values.
This method demands lots of work to be made by the modeller, by creating the matrices by hand, before a scene can be used.

Although not about snow Hsu and Wong [1995] presented a method for dust accumulation in their "Visual Simulation of Dust Accumulation". In this paper they use an exposure function to tell whether the surface should be covered in dust or not. The exposure function is calculated by sampling the surrounding area with rays to find any occluders.

A variant of this method turned out to be a viable way to handle the process of snow accumulation. The method we will present uses [Hsu and Wong 1995] as a basic foundation.

3. Snow Accumulation

The process of rendering accumulated snow can be split into two parts. The first part is to decide what regions should receive snow. The second is to actually render the snow in a convincing way at the places decided by the first part of the algorithm.
To accomplish the first step a function called the Snow Accumulation Prediction Function is introduced. This function should take a point in space and calculate how much snow that point has received. Factors that should be taken into account are surface inclination and exposure to the sky.

3.1. Snow Accumulation Prediction Function

Due to gravity a surface facing upwards should receive more snow than a vertical surface. However, even a horizontal surface does not accumulate any snow if it is occluded from the perspective of the sky. It seems like the task of calculating the prediction function can also be separated into two mutually independent parts. One part should calculate the snow contribution due to the inclination of the surface and another part should calculate the effect of occlusion.
Let us formulate a Prediction function in terms of these two parts:

$$f_p(p) = f_e(p) \cdot f_{inc}(p)$$

where p is the point of interest
f_p is the prediction function
f_e is the exposure component
f_{inc} is the component giving the snow contribution due to inclination

The exposure part (f_e) should vary between 0 and 1 indicating the amount of occlusion that would prevent snow from falling on the surface. This value should vary gracefully to achieve a smooth transition from an area with snow to an occluded area. Hsu and Wong [1995] calculates this value by sampling the surrounding area with rays uniformly distributed over the upper hemisphere in order to search for potential occluders. Unfortunately this is not a viable way of doing it if we want to calculate this in real-time without lots of pre-processing for all surfaces. An alternative way of implementing this function based on the shadow mapping technique will be examined later in this paper.

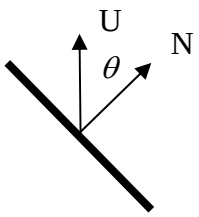


Figure 1: Inclined surface

The f_{inc} should work in a similar way to the Dust Amount Prediction Function in [Hsu and Wong 1995]. A surface facing towards the snow direction should receive more snow than a surface facing away from it. The amount should depend on the angle, with a rather step falloff when the angle between the normal and a vector pointing upwards grows (Figure 1). Surfaces facing away from the sky should not receive any snow. This implies that the function should decrease from 1, when the angle is 0, to 0 when the angle is 90 degrees. If the angle is greater than 90 f_{inc} should be 0.
This gives a function like this:

$$f_{inc} = \begin{cases} \cos\theta + n, & 0 \le \theta \le 90 \\ 0, & \theta \ge 90 \end{cases}$$

where θ is the angle between vector N and U
n is a small positive noise value

A small noise value [Perlin 1985], typically between 0 and 0.4, is used to get a more natural look, and to account for the phenomena called flake dusting [Fearing 2000], where snow dust clings to a steep uneven surface.

3.2. Snow Colour Function

After the snow accumulation of a certain surface is determined it's time to calculate the correct shading of the snow. The light model that is used is the normal Phong illumination model.
To get the typical look of snow a noise function is used to distort the normal of the surface. This way a realistic approximation of the look of a snow cover is achieved. To get the glittering effect of snow the normal is distorted slightly more for the specular part of the light calculation than for the diffuse part. The derivative of the exposure function is also used to transform the normal to get an illusion of actual snow depth around any occlusion boundaries that may exist in the scene. In the implementation the normals are calculated in this way:

$$N_\alpha = N + \alpha n - dE$$

where N is the original normal
α is a scalar value indicating how much distortion should be applied
n is a normalized vector containing three noise values
dE is a vector containing a scaled value of the exposure derivatives in respective direction of screen space

A distorted value α of 0.4 was used in the images presented in this paper. This was chosen through testing different values and deciding on what looked best.
The renormalized resulting normal is then used in the diffuse part of the lighting equation. For the specular part another distortion term was added with the α value 0.8 to get a more glittering effect on the snow.
To calculate dE the derivative of the exposure function in both x and y direction of the screen is needed. In this implementation the derivative operations of Cg is used. The derivative with respect to x is placed as the x-component and the derivative with respect to y in the z-component, assuming that the y-axis points upwards. The resulting vector is then scaled to obtain a suitable impression of decreasing height when the exposure function decreases.
When calculating the snow colour as above a white colour should be used in the Phong equation. The blue part of the colour should

be slightly higher than the rest to give the snow a more glittering effect.

3.3. Full Snow Equation

The full equation to calculate the colour of accumulated snow then becomes

$$C = f_p \cdot C_s + (1 - f_p) \cdot C_n$$

where C_s is the snow colour calculated with the distorted normal, as explained above
C_n is the surface colour without snow

To obtain an impression of thickness to the snow each vertex should be displaced depending on the f_p value. The amount to displace the vertices depends on the scene and needs to be tested to achieve the best result. The displacement introduces the restraint on the geometry that it should be closed to give the impression of actual height of the snow.

4. Implementation

The above algorithm where implemented in a pair of vertex/pixel shaders. Most of the equations can be implemented in a straight forward way when implementing the standard shading equations, as described by [Everitt et al. 2002].

4.1. Implementing the exposure function

The exposure function did not lend itself to the same easy implementation as the other components of the equation. To calculate the exposure of a surface, knowledge about the surrounding world is needed. This does not suit well with the limitations inherent when working with the pixel shader on a modern GPU. To solve this, a solution is taken from the way shadows are implemented with the aid of a depth buffer. This image space algorithm is very suitable for what we are trying to do here.
As a pre-processing stage the whole scene is rendered with respect to the sky, using a parallel projection. The depth buffer is then saved for later usage.

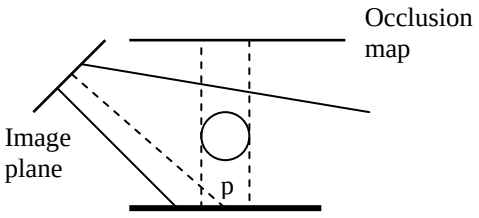


Figure 2: Point p is not the closest point to the sky, it will not be covered b snow

When rendering the scene each fragment is projected into the sky view frustum, as described in [Everitt et al. 2002], and compared with the stored depth value in the occlusion map. If the fragment is further away from the sky than the value indicated in the occlusion map, as in Figure 2, no snow should be drawn, otherwise snow should be drawn. This method works but is still unsatisfactory because it produces very sharp and sudden occlusion boundaries, as can be seen in the picture below.

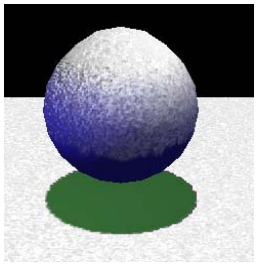


Figure 3: One sampling from the occlusion map produces very sharp transitions.

To address this issue the depth map is sampled a number of times, with the first sample on the original fragment position and the others in a circle around it in the occlusion map. The number of snow covered fragment is then divided with the total number of samples to get a fractional value for the occlusion. This creates a better result, but it still leaves us with plateaus in the snow and possible artefacts on objects as seen in Figure 4. The length of the offset used for the different samples determines how big the occlusion boundary becomes. A larger offset produce a smoother occlusion boundary. The larger the occlusion boundary should be the more samples need to be done to get enough overlapping in the samplings. How big the offset should be is entirely dependent on the size of the scene the occlusion map covers, and the desired area of the occlusion boundary.
To sample the depth buffer in the area surrounding the current fragment the change in z direction due to offsets in the depth map must be known. To get this value the derivative functions in cg are used to calculate the proper position of neighbouring fragment.
All of the above mentioned offsets are performed in the projected space of the occlusion map.

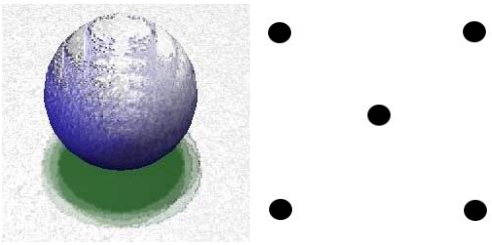


Figure 4: (a) 5 Samplings produces better transition but introduces artefacts. (b) Sampling pattern

As can be seen this produces quite a bit of artefacts in the exposure value. To fix this the final result is combined with a noise value in the range of [0, 0.5], if it is bigger than 0, to produce a more natural looking boundary, and to conceal surface artefacts due to discontinuities in the surface.

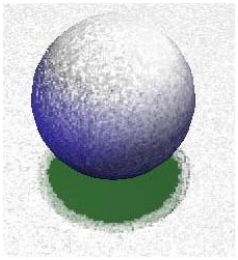


Figure 5: Same scene as figure 3 but with noise included.

The adding of the noise value effectively removes all of the artefacts shown in figure 3. It also makes the occlusion boundary look much more natural.

4.1.1. Vertex Displacement Method 1

One problem with this method is that the vertex program can't read from the occlusion texture. This means that we have to provide the snow values to the vertex program in some other way. One way is to introduce another pre-processing stage where the exposure map is first read back from the GPU. The position of each vertex in the exposure map must then be calculated. This is done by scaling the x and z coordinate according to the world size so that they can be used to index into the occlusion map, ignoring the y coordinate. This value should then be streamed to the vertex shader in the same way as position and normals.

4.1.2. Vertex Displacement Method 2

Another way of solving the problem with the vertex shaders inability to access the occlusion map is to split the rendering into 2 passes. The first pass should draw the scene without snow and the second pass with the snow. When displacing the vertices for the snow pass no consideration should be taken to if the vertex is snow covered or not, only the inclination should be considered. The exposure value should instead be assigned as an alpha value to each fragment in the fragment shader. The second pass should then be blended onto the first, using alpha as blending factor for the second pass and one minus alpha for the first. This method is easier to implement than the first, and places less demands on the format of the scene. It does unfortunately create some artefacts where the snow floats above the scene that can be annoying in certain scenes where the user can view the scene from any perspective. Figure 6 show this artefact.

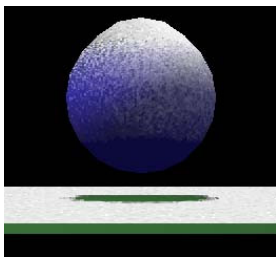


Figure 6: Floating snow artefact seen on the ground. The snow layer floating some distance above the ground

The artefact is extremely visible in the above picture because of the fact that the ground plane isn't a closed geometry. But even when working with closed geometries artefacts as the above can still happen, especially with very sharp corners.
Method one does completely eliminate this artefact but introduces the need for more pre-processing and the need to store the values which might be a problem when using already existing scenes and scene formats.

4.2. Implementing noise

All of the noise values used in this paper can be implemented in the fragment shader by using a 3D texture filled with random 3 component values in the range [0, 1]. This texture is then used to draw random numbers that will always be the same for the same input.
To get the correct noise appearance the random texture can be sampled a number of times, called octaves [Perlin 1985], each octave having double frequency of the last, and half the amplitude. The sum of these octaves can then be used for a noise value.
The noise used in the pictures shown in this paper where created by reading 3 octaves from the 3D texture. The single noise value used in f_{inc} was calculated by adding the 3 components together and the dividing by 3 to bring the range back to [0, 1]. The noise vector read from the 3D texture was then expanded to the range [-1, 1] and normalized to create a suitable vector for distortion of the normals used in the light calculation.

4.3. Performance Issues

The implementation of the algorithm includes a lot of normalizations that affects the performance quiet thoroughly. To avoid this a cube map is used for normalization purposes.
In a cube map only the direction of a vector is used for lookup, not the magnitude. This can be used for normalizations by storing a normalized version of the direction vector in each component of the cube map. The components of the normalized vectors must first be transformed to the range [0, 1] before they are stored in the cube map. This is done by multiplying the vector by 0.5, taking the elements into the range [-0.5, 0.5]. By then adding 0.5 the range [0, 1] is achieved. The components are then encoded in the RGB components of the cube map texture. To use the cube map for normalization a texture lookup should be done as usual, with a vector as texture coordinates. The resulting vector must then be unpacked from [0, 1], the range in which colours are normally stored, to [-1, 1] by multiplying by 2 and subtracting 1.
By using linear interpolation on the texture lookup the cube map turns into a smooth normalization function without needing very high resolution. A cube map of 64x64 or 128x128 is usually enough.

5. Performance

The implementation is tested on a machine with a Geforce FX 5600 Ultra, using CG to compile the shaders to the NV_vertex_program2 and NV_fragment_program.
The performance is directly dependent on the resolution and the amount of the screen covered with potential snow covered surfaces.
In the demo presenting two instances of the Stanford bunny, Figure 7, each consisting of 16000 triangles an average frame rate of 13 frames per second were achieved. This was with a screen resolution of 600 x 600. When increasing the screen size to 900 x 900 the frame rate dropped 11 frames per second.

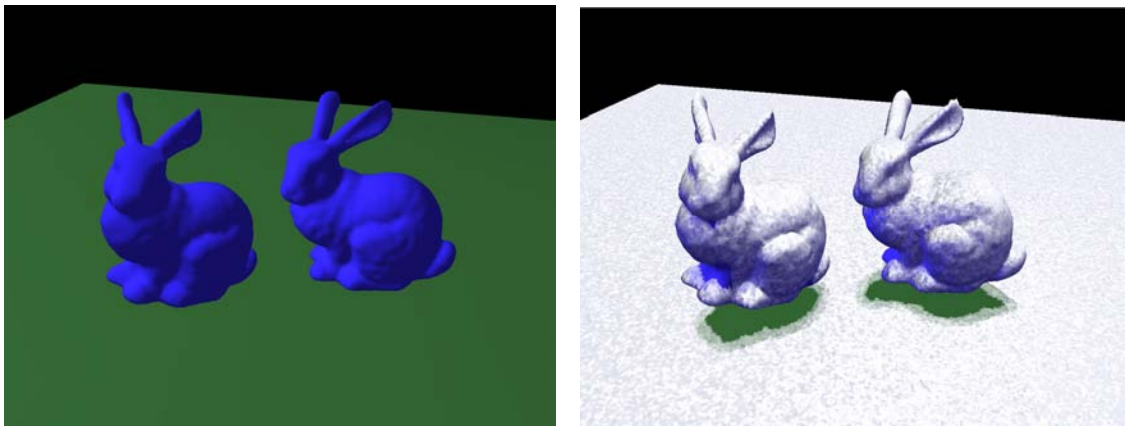


Figure 7: Stanford bunny without and with snow

6. Discussion

One problem with the snow accumulated in the way proposed in this paper is that it does not take into account the area of the region being covered in snow. This means that small parts, as the small twigs in the tree pictures in Figure 8 and Figure 9, can be loaded with an unnatural amount of snow. There isn't really much to do about this, because it would demand additional knowledge about the scene that does not exist at the fragment level.
The tessellation of the snow cover will be that of the underlying geometry. This leaves the scene modeller free do add polygons in places where a smoother curve is desired.
There are several possible improvements that could make the proposed method faster. One is to save the four surrounding sample values of the occlusion map in the RGBA parts of a new texture to reduce the numbers of samples needed to two, non dependent lookups. This would be done as a pre-processing step when the occlusion map is created.
Something that speaks for this method in the future is that it is independent of the number of objects in the scene, enabling the usage of complex and cluttered scenes without any extra work. And as the speed of fragment processing increases with the advantage of GPU, speed would cease to be an issue.
Another problem with the method is if you want to have dynamic objects acquire snow as they move out in the scene. That would mean that when they move in to shelter the snow accumulated would be lost on them.

7. Conclusion

In this paper, a new method for calculating snow on a fragment level was presented. The method uses a depth map to find out what parts of the scene should be covered with snow. The snow is then calculated per fragment in the scene without needing any more pre-processing of the scene data. Although the method isn't very fast as of today, increases in the computational power of today's graphic hardware should make this method a good candidate for snow accumulation and rendering in the future. The advantage of not needing to modify or store any extra data about the scene except for the occlusion map means that it should be easy to implement and combine with other existing techniques for rendering.

8. Future Work

Something that would be an interesting extension to the above presented method is the usage of a control map where things as footsteps could be drawn. This could be done by using another texture map stretched in the same way as the occlusion map. Each fragment would then multiply its exposure value with the value in the control map. Footsteps could then be drawn in the control map as half occluded fragments.
Another area that could prove interesting is the possibility to take into account the influence of wind on the falling snow. Perhaps this could be modelled by tilting the projection when calculating the occlusion map.

References

Everitt, C., Rege, A., and Cebenoyan, C., 2002. Hardware shadow mapping, ACM SIGGRAPH 2002 Tutorial Course #31: Interactive Geometric Computations using graphics hardware, ACM, F38-F51

Fearing, Paul 2000. Computer Modelling Of Fallen Snow. Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 37-46

Haglund, H., Anderson, M., and Hast, A., 2002. Snow Accumulation in Real-Time, Proceedings of SIGRAD 2002, 11-15

Hsu, S.C, and Wong, T.T. 1995. Visual Simulation of Dust Accumulation, IEEE Computer Graphics and Applications 15, 1, 18-22

Nishita T., Iwasaki, H., Dobashi, Y., and Nakamae, F. 1997. A Modeling and Rendering Method for Snow by Using Metaballs. Computer Graphics Forum, Vol 16, No. 3, C357

Perlin, Ken, 1985. An Image Synthesizer, Computer Graphics (Proceedings of ACM SIGGRAPH 85), 19, 3, 287-296

Summers, Robert W., O'Brien, James F., and Hodgins, Jessica K. 1998. Animating Sand, Mud, and Snow. The Proceedings of Graphics Interface'98, 125-132

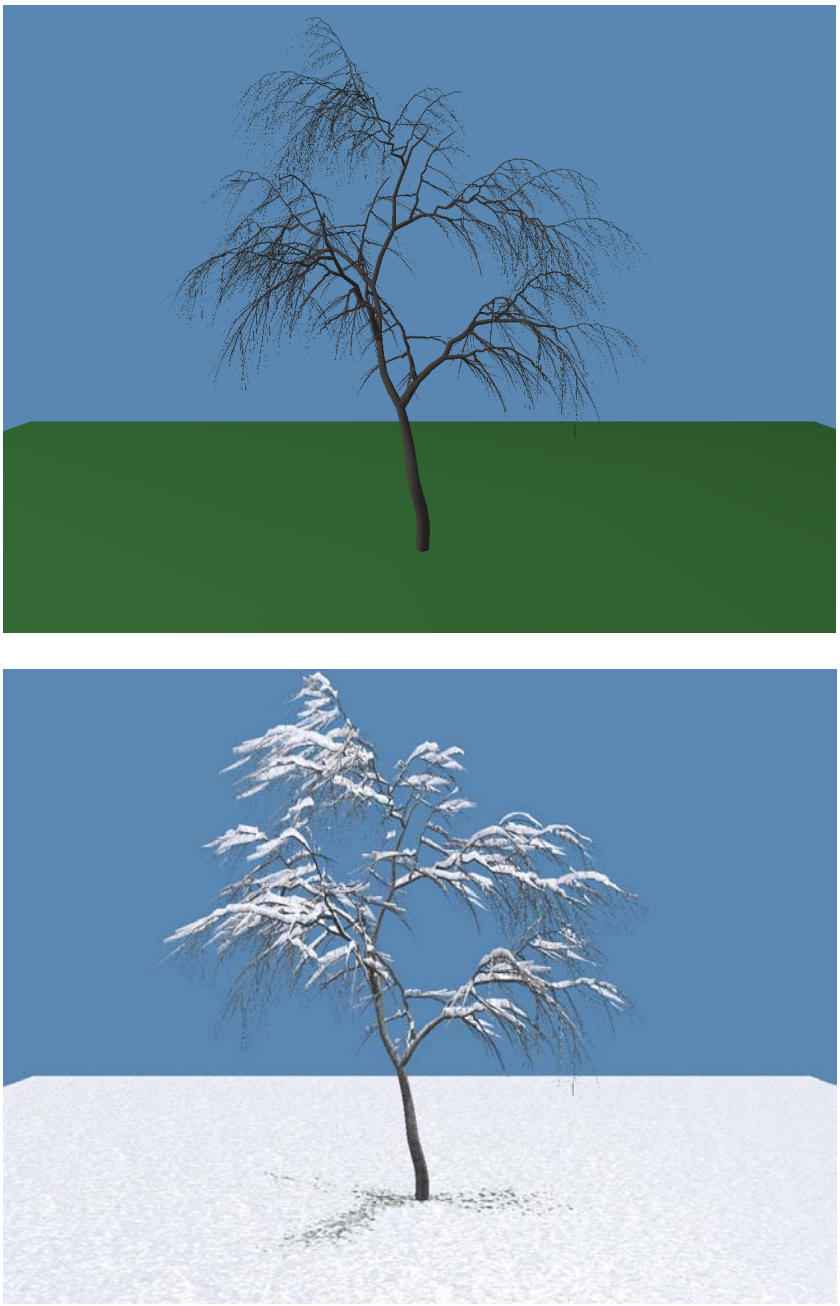


Figure 9: The algorithm tested on a more complex scene containing lots of small structures. As can bee seen in the picture the algorithm does not take into account the area of the surface being displaced. This results in the small twigs getting more snow than would look natural.

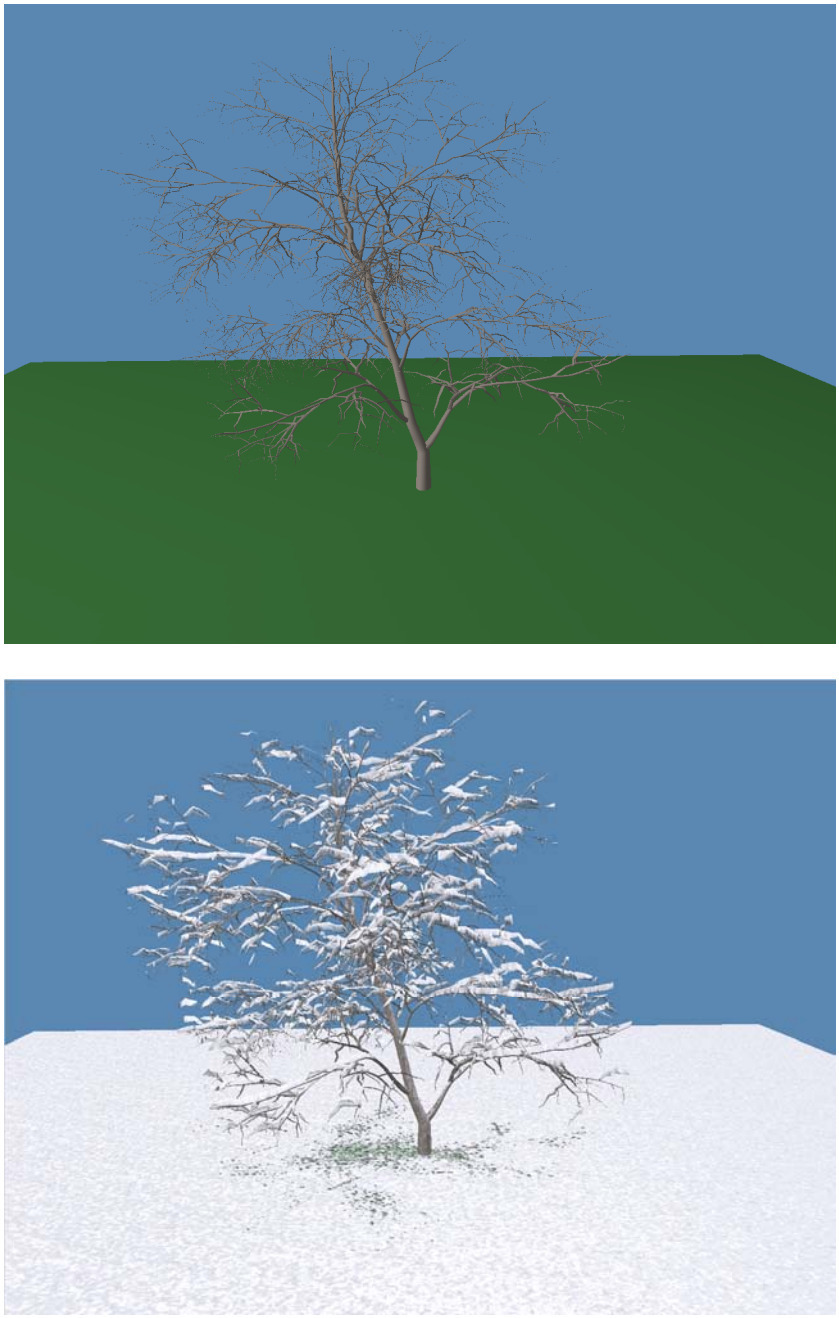
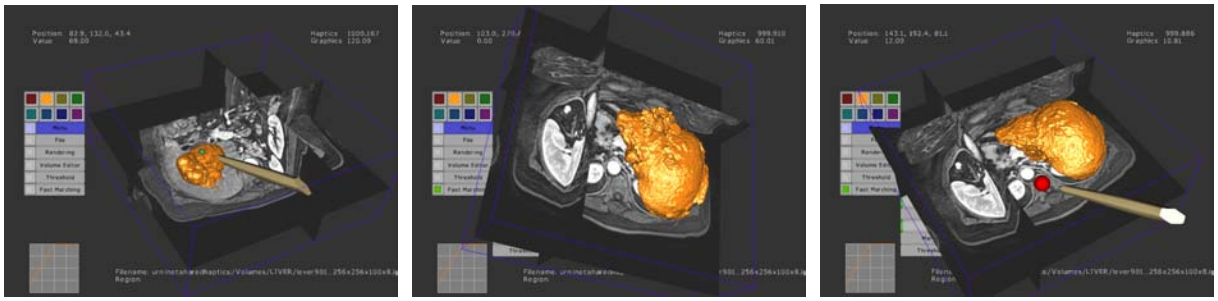


Figure 10: Another tree used for testing of the snow algorithm. As can be seen the area beneath the tree is partially covered depending on amount of branches above it.

Fast surface rendering for interactive medical image segmentation with haptic feedback

Erik Vidholm* Jonas Agmund†
Centre for Image Analysis
Uppsala University



(a) Initialization of a seed-region inside the liver.

(b) Initial segmentation result that contains artifacts due to the low contrast between the liver and the surrounding tissues.

(c) Result after editing.

Figure 1: Screenshots from the interactive segmentation environment.

Abstract

In this work, we present a haptic-enabled application for interactive editing in medical image segmentation. We use a fast surface rendering algorithm to display the different segmented objects, and we apply a proxy-based volume haptics algorithm to be able to touch and edit these objects at interactive rates. As an application example, we show how the system can be used to initialize a fast marching segmentation algorithm for extracting the liver in magnetic resonance (MR) images and then edit the result if it is incorrect.

CR Categories: I.3.6 [Computer graphics]: Methodology and techniques—Graphics data structures and data types, Interaction techniques. I.4.6 [Image processing and computer vision]: Segmentation.

Keywords: marching cubes, surface tracking, volume haptics

1 Introduction

Image segmentation is the task of finding a certain object in an image and label all the voxels inside the object as foreground, and all other voxels as background. In medical segmentation, objects should be extracted from different data sets obtained through, e.g., Computed Tomography (CT) or Magnetic Resonance Imaging (MRI). An object to be segmented could typically be a part of the brain or the liver. Even though many methods have been proposed for automatic segmentation, it is still seen as an unsolved problem since the methods are not general enough. In semi-automatic methods, some degree of manual interaction is involved to improve the result. Ideally, the user should give an initialization to the algorithm and then examine the final result and if necessary edit it. The efficiency of this interactive part is highly dependent on the quality of the user interface. The user needs to be provided with proper tools for the specific task, and the learning threshold should not be too high. When working with volume images it is a huge step just to map interaction in 2D to events in 3D. By using more advanced input devices combined with different depth cues (e.g., stereo), it is possible to overcome this problem.

Interactive editing and manipulation of volume data for design and modeling purposes has been referred to as *sculpting* by previous authors. In general, a sculpting system consists of a set of modeling tools together with fast surface rendering and/or haptic rendering algorithms for data display. In [Galyean and Hughes 1991], a sculpting system with various free-form tools was developed. An octree-based system was proposed in [Bærentzen 1998] where "spray-tools" and constructive solid geometry (CSG) tools were used. The use of haptic feedback in volume sculpting was suggested already in [Galyean and Hughes 1991], but realized first in the work described in [Avila and Sobierajski 1996]. In the recent paper [Kim et al. 2004], a combined geometric/implicit surface representation is used along with tools for haptic painting based on texture techniques. The connection between haptic volume sculpting and interactive volume image segmentation is close, but not much work has been done in this area. Haptic interaction was used by [Harders and Székely 2002] for centerline extraction during segmentation of

*e-mail: erik.vidholm@cb.uu.se
†e-mail: jonas.agmund.4911@student.uu.se

tubular structures, and in [Vidholm et al. 2004] haptic feedback was used to facilitate the placement of seed-points in MR angiography data sets for vessel segmentation. Examples of non-haptic interactive segmentation tools for volume images that have inspired our work are found in [Kang et al. 2004].

In this paper, we propose the use of editing tools based on morphological image processing operators in combination with haptic feedback, stereo graphics, and a fast surface rendering algorithm to interactively edit and manipulate segmented data. Haptics provides the possibility of simultaneous exploration and manipulation of data. In our work, realistic feedback is not the most important issue. More important for us is that the user works more efficiently with guidance by haptic feedback than without. The aim is to considerably reduce the amount of user time required in the segmentation process.

The paper is organized as follows: In Section 2, we give an overview of our visuo-haptic environment for interactive segmentation. A brief description of the volume visualization based on 3D texture mapping is given in Section 3. In Section 4, we present the fast surface renderer and some implementation issues. Section 5 gives an introduction to volume haptics and describes how we use haptic feedback for editing. An example application is given in Section 6 and in Section 7 we present our results. Finally, we summarize the paper with conclusions and future work in 8.

2 System overview

In this section, we give an overview of our environment and the interactive segmentation application.

2.1 Hardware and software

Our setup consists of a Reachin desktop display [Thurfjell et al. 2002] which combines a 3 degrees of freedom (DOF) PHANToM desktop haptic device with a stereo capable monitor and a semi-transparent mirror to co-locate graphics and haptics. See Figure 2. The workstation we use is equipped with dual 2.4 GHz Pentium 4 processors and 1GB of RAM. The graphics card is a NVidia Quadro 900XGL with 128MB of RAM. For stereo display, Crystal Eyes shutter glasses are used. The software has been implemented in the Reachin API, a C++ API that combines graphics and haptics in a scene-graph environment based on the VRML97 standard.

Figure 2: The Reachin desktop display.

2.2 Interactive segmentation

Most of the user interaction is performed with the PHANToM device through 3D widgets and volume editing tools. A Magellan space mouse is used for additional input. The haptic/graphic user interface is used for interaction with the main parts of the system, i.e., the 3D texture mapper, the image processor, the volume editor, the surface renderer, and the haptic renderer. All of these share access to a volume image $\mathbf{V}$ that we want to extract objects from. This image is typically obtained through MRI or CT. In the 3D texture mapper, we visualize the data in $\mathbf{V}$ by utilizing the hardware accelerated 3D texture mapping features of the graphics card. The image processor contains a set of different segmentation algorithms that has $\mathbf{V}$ as input and produce segmented volumes $\mathbf{S}$ as output. A segmented volume $\mathbf{S}$ is integer valued, and can contain several objects labeled between 1 and N, where N is the number of objects. Object no. j consists of the voxels $\Omega_j = \{\mathbf{x}|\mathbf{S}(\mathbf{x}) = j\}$. The background is labeled 0. In the surface renderer, fast surface reconstruction of the segmented objects in $\mathbf{S}$ is performed. The haptic renderer computes forces based on the data in $\mathbf{S}$, and is closely connected to the volume editor which contains various editing tools. As an option, the haptic feedback can also be based on $\mathbf{V}$ for enhanced navigation. Figure 3 illustrates the structure of the system.

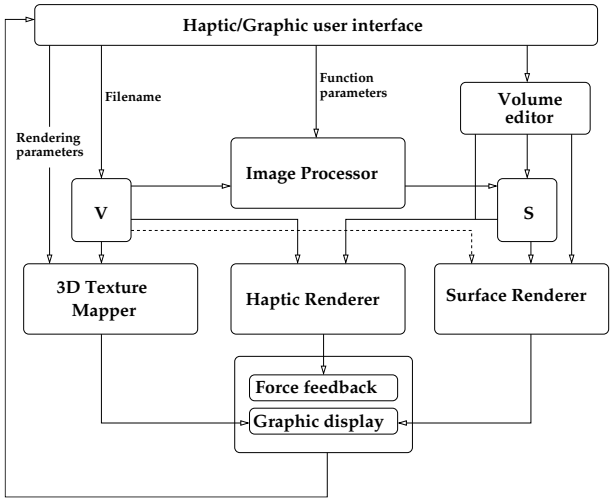


Figure 3: An overview of the interactive segmentation system.

3 3D texture mapping

Two different volume visualizations are used in our system. The surface rendering algorithm described in Section 4 is used to display segmentation results, while the original (medical) volume images are visualized through 3D texture mapping. The basic idea is to regard the whole volume image $\mathbf{V}$ as a texture map defined over $[0,1]^3$, and the texture mapping as the interpolation of the values in this domain. The default visualization in our application is a multiplanar reconstruction (MPR) consisting of three texture mapped or-

thogonal slice planes that can be moved along the corresponding coordinate axis. It is also possible to view maximum intensity projections (MIPs) of the data. We construct the MIPs by mapping the 3D texture onto a stack of view-plane aligned polygons that are rendered in back-to-front order and blended together.

To adjust contrast and brightness in these different projections, we use texture shading and the register combiner features of the graphics card. Two textures are loaded into texture memory: the volume **V** and a 1D texture **CB** that we use to store the contrast/brightness transfer function. By re-programming the register combiners, we can use the texture value from **V** as a texture coordinate for **CB**, and use that value in the rasterization. The same technique can also be used in ordinary volume rendering to implement transfer functions for opacity and color. Since the texture shading and register combinations are performed before the blending operations, any contrast/brightness adjustment affects both the slice planes and the MIP.

4 Surface rendering

A common way to use surface rendering of volume images is iso-surface extraction, i.e., a surface along which the volume image is equal to a certain threshold value, or iso-value. Interpolation is often used to achieve a smoother surface, and also shading where the surface normals are based on the volume gradient. Iso-surface extraction algorithms can be based on ray-casting methods or polygonization like in [Wyvill et al. 1986] and the more well-known marching cubes (MC) algorithm [Lorensen and Cline 1987]. We have chosen the MC algorithm since it is straight-forward and fits well into the already existing visualization environment.

In our application, we want to render the segmented and labeled objects contained in **S**. This is done by using the label of each object as iso-value.

4.1 Surface detection

The first step in the MC algorithm is to identify the cells in the volume that are intersected by the iso-surface. A cell is a cube consisting of eight $(2 \times 2 \times 2)$ neighboring voxels.

In the original implementation, the whole volume is traversed and all cells are examined for surface intersection. This is very inefficient if the surface only intersects a small part of the cells in the volume, which usually is the case.

One way to speed up the surface extraction is to use alternative data representations of the volume instead of an ordinary 3D array, e.g., an octree [Wilhelms and van Gelder 1992; Bærentzen 1998]. Drawbacks of using octrees are that the tree needs to be re-generated when the image is manipulated, and it is not straight-forward to make use of shared vertices and normals during the triangle generation.

To facilitate image manipulation and sharing of vertices and normals, we decided to use an ordinary 3D array representation as in [Galyean and Hughes 1991]. To avoid the traversal of non-intersecting cells, surface tracking [Shekhar et al. 1996] is used. This method takes advantage of surface connectivity. Given a seed-cell, i.e., a cell in the volume which is intersected by the surface, the surface is visited one cell at a time by following the connectivity until all connected cells have been visited. The connectivity for a certain MC configuration can be pre-computed and stored in a lookup-table (LUT) for efficient tracking, see Section 4.3.2.

4.2 Triangle generation

Once the surface is detected, each cell intersected by the surface should be triangulated according to the MC configurations. There are several options to consider when creating the triangles. Each vertex position of a triangle can be interpolated to give a more accurate position of the surface, or it can simply be set to a midway position on each cell edge. When dealing with binary data as in our case, no interpolation is necessary since it will default to the midway position. Regarding the normals, they can be calculated by either using the geometric normal of the triangle, or by using the gradient in the volume image. If the gradient is used in conjunction with interpolation of vertices, the gradient needs to be interpolated too. Computation of gradients and interpolation of positions are time-consuming and should be avoided unless needed for accurate visualization purposes.

One of the major drawbacks with MC is the excessive output of triangles. Since each cell intersected by the surface can result in up to five triangles, even a small volume image can result in surfaces of a massive number of polygons. Different algorithms for triangle decimation have been proposed [Montani et al. 1994; Shekhar et al. 1996],

4.3 Implementational aspects

The following was taken into consideration when implementing the MC-based surface renderer:

- The volume **S** should be easy to access and manipulate for the surface renderer, the haptic renderer, and the image processor.
- The renderer should be optimized for extraction and rendering of segmented data, but interpolation of vertices and gradient based normal computations should be included as an option.
- When the volume is manipulated, re-rendering of the surface must be efficient.

More details can be found in [Agmund 2004].

4.3.1 Data structures

The following data structures are used by the surface renderer:

- The original volume **V** of size $W \times H \times D$.
- The segmented volume **S** of the same size as V.
- A cell index array **C** of size $(W-1) \times (H-1) \times (D-1)$ containing the MC configuration index for each cell. Cells that are intersected by the surface has an index between 1 and 254 and the non-intersected cells have index 0 or 255.
- Two 2D coverage arrays $\mathbf{X}^-$ and $\mathbf{X}^+$ of size $(H-1) \times (D-1)$ containing minimum and maximum x-coordinates for surface intersected cells in each line in **C**. A value of zero in $\mathbf{X}^+$ means that the surface does not intersect any cell on the current line.
- A vertex list $\mathbf{v}_l$ for storing vertex positions.
- A normal list $\mathbf{n}_l$ for storing vertex normals.
- An index list $\mathbf{i}_l$ for triangle generation from $\mathbf{v}_l$ and $\mathbf{n}_l$.
- Three index cache arrays $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3$ used for caching already computed indices during the triangle generation.

The main steps in the implementation of our surface renderer is shown in Figure 4.

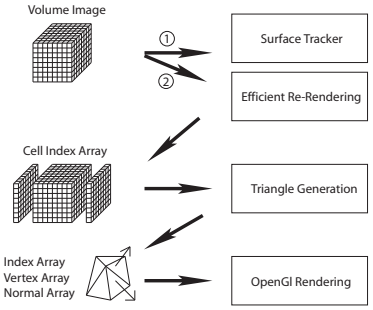


Figure 4: Overview of the surface renderer.

4.3.2 Surface tracking

The surface tracking can be started immediately if an intersected seed-cell is known and the surface to be extracted is connected. In cases where this is not true, the whole volume is scanned to find all existing surfaces. The basic algorithm is as follows: First, **C** is cleared and set to zero. $\mathbf{X}^-$ is set to W and $\mathbf{X}^+$ is set to 0. A linear search through **S** is performed until a given iso-value is found (a simple equality test). If an index is found and the cell is not previously visited (stored in **C**), surface tracking is started at the seed-cell. This procedure is repeated until the whole volume is traversed.

The surface tracking uses the values in **C** to keep track of already visited cells, and the pre-calculated connectivity LUT to find in which directions the surface is connected. See Figure 5. To be able

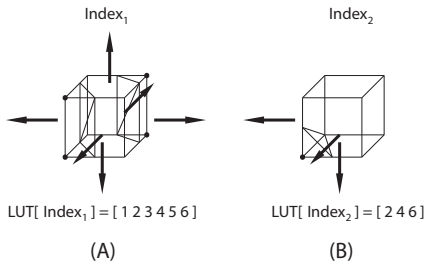


Figure 5: Example of information in the connectivity LUT for two MC configurations with connectivity in 6 directions (A) and in 3 directions (B).

to know in which order and which cells to visit a *deque* is used. A deque is a modified linked list, being efficient when elements are to be added and removed only to the end and beginning of the list. The algorithm is initialized by putting the seed-cell in the deque. The algorithm continues in the following way:

1. Pop the first cell in the deque.
2. Calculate the MC index of the current cell and insert the index into **C** for future use in the triangulation and to mark the cell as visited.
3. Compare the current x-position with the values in $\mathbf{X}^-$ and $\mathbf{X}^+$ and update if necessary.
4. Use the connectivity LUT to determine which directions the surface continues in and put these cells at the end of the deque, if they have not already been visited.
5. Repeat from step 1 until the deque is empty.

4.3.3 Vertex and normal computations

When all surfaces are found, the triangulation is performed. This is a separate process using the information stored in **C** and the coverage arrays $\mathbf{X}^-$ and $\mathbf{X}^+$. The original volume **V** is not used here, unless if interpolation is performed or if gradient-based normals are used.

The triangle generation is performed through an *xyz*-order traversal of **C**. During this process, the coverage arrays are consulted to skip the first and last non-intersected cells on each line, respectively. A LUT stores which edges for each MC configuration that will contribute with a triangle vertex. Due to the scan direction, there are only 3 of the 12 cell edges that can contribute with a new vertex. For vertex and normal sharing between triangles, the cache arrays store indices of already computed vertices and normals. This is illustrated in Figure 6. Vertex normals can be calculated in two

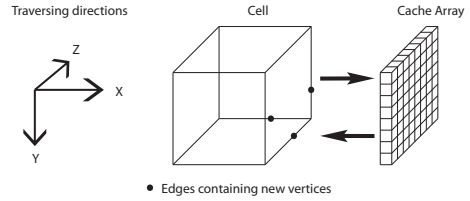


Figure 6: Illustration of the three edges that contribute with new vertices. Information about vertices on the other nine edges are already known and can be retrieved from the cache arrays.

different ways. The first, which is the most efficient, uses the average geometrical normal from each triangle that shares the vertex. The efficiency lies in that the geometrical normals for each triangle are pre-calculated and stored in a LUT for each MC configuration. This is possible since we work with binary data and only use the midway position on each edge. The second, more costly, method to calculate normals is to use the gradients from **V** at each vertex position.

4.3.4 Efficient re-rendering

In the application, there are two ways of manipulating the segmented volume **S**. The first is to apply a global method (e.g., thresholding of **V**) that requires a total update according to the surface tracking algorithm. The second is to apply local editing operations from the volume editor. Since these operations only affects a small part of the image it is enough to traverse a sub-volume which extent depends on the current editing tool, and to update the corresponding values in **C**, $\mathbf{X}^+$, and $\mathbf{X}^-$. In the current implementation, we must re-generate all triangles, but since all computations are reduced to table lookups this is not a problem. However, in the future we will try to improve the implementation to modify only parts of $\mathbf{v}_l$, $\mathbf{n}_l$, and $\mathbf{i}_l$.

5 Interactive editing with haptic feedback

One of the first attempts to use haptics for the display of volume data was made in [Avila and Sobierajski 1996]. In their work, the force feedback provided to the user is a direct mapping from the position of the haptic probe to a force computed from intensity values and local gradients at that position. A drawback with this type of method is instability. The rendering parameters can be hard to tune correctly in order to avoid force oscillations. In surface haptics, the stability problem was solved by introducing a virtual *proxy* that is connected to the haptic probe through a spring-damper [Ruspini et al. 1997].

5.1 Proxy-based volume haptics

The idea in proxy-based haptic rendering is to constrain the proxy to certain movements in a local reference frame (LRF) and to provide a resulting force vector proportional to the displacement of the haptic probe relative to the proxy. Proxy-based rendering of volumetric data was first proposed by [Lundin et al. 2002], where a LRF for scalar volumes is obtained through tri-linear interpolation of the volume gradient at the proxy position. The gradient is used as a surface normal that defines a surface to which the proxy is constrained. It is also shown how friction and viscosity can be rendered and how different material properties can be simulated by using haptic transfer functions. In [Ikits et al. 2003], a framework for more general LRFs and proxy *motion rules* was presented.

5.2 Haptic feedback when editing

We have based our haptic rendering on the two works mentioned in Section 5.1 combined with the idea of a tool with sample points on the surface [Petersik et al. 2003].

The basic steps in the haptic loop are as follows: let $\{\boldsymbol{e}_0, \boldsymbol{e}_1, \boldsymbol{e}_2\}$ denote the LRF, $\boldsymbol{p}^q$ the proxy position at time step q, $\boldsymbol{x}^q$ the probe position, and $\boldsymbol{d} = (\boldsymbol{x}^q - \boldsymbol{p}^{q-1})$ the displacement of the probe relative to the previous proxy position. In each iteration of the haptic loop the proxy is moved in small steps according to user input and rendering parameters such as stiffness and friction. Allowed proxy movements are defined by certain motion rules for each axis in the LRF. The proxy position at time step q is computed as

$$\boldsymbol{p}^q = \boldsymbol{p}^{q-1} + \sum_{i=0}^{2} \Delta p_i \boldsymbol{e}_i,$$

where Δp_i is a motion rule function of the displacement $d_i = \boldsymbol{d} \cdot \boldsymbol{e}_i$. The resulting force is computed as $\boldsymbol{f}^q = -k(\boldsymbol{x}^q - \boldsymbol{p}^q)$, where k is a stiffness constant.

We use a spherical tool with radius r that is centered at $\boldsymbol{p}$. In a precomputed array we store uniformly spaced sample points $\boldsymbol{t}_i$, $||\boldsymbol{t}_i|| = 1$, so that the points $\boldsymbol{T}_i = \boldsymbol{p} + r \cdot \boldsymbol{t}_i$ are located on the tool surface. The sample points that are in contact with the current object are used to define the normal component $\boldsymbol{e}_0$ in our LRF:

$$\boldsymbol{e}_0 = -\frac{\sum_{i \in I} \boldsymbol{t}_i}{||\sum_{i \in I} \boldsymbol{t}_i||},$$

where $I = \{i | \mathbf{S}(\boldsymbol{T}_i) > 0\}$. The tangential direction $\boldsymbol{e}_1$ is constructed by projecting $\boldsymbol{d}$ onto the plane defined by $\boldsymbol{e}_0$ [Lundin et al. 2002]:

$$\boldsymbol{e}_1 = \frac{\boldsymbol{d} - (\boldsymbol{d} \cdot \boldsymbol{e}_0)\boldsymbol{e}_0}{||\boldsymbol{d} - (\boldsymbol{d} \cdot \boldsymbol{e}_0)\boldsymbol{e}_0||}.$$

Since $\boldsymbol{e}_1$ is constructed in this way, the third component of the LRF is not needed, but it can easily be computed as $\boldsymbol{e}_2 = \boldsymbol{e}_0 \times \boldsymbol{e}_1$.

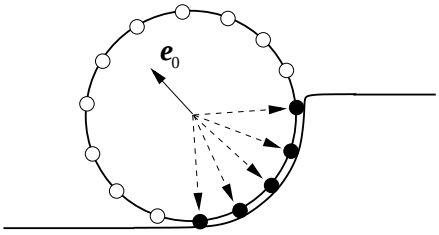
The motion rule for the normal direction $\boldsymbol{e}_0$ is

$$\Delta p_0 = \begin{cases} d_0 & \text{if} \quad d_0 > 0 \\ -\max(|d_0| - T_0/k, 0) & \text{if} \quad d_0 \leq 0 \end{cases},$$

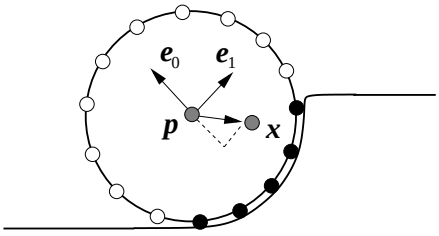
where the threshold T_0 is the force the user must apply to penetrate the surface with the tool. For the tangential direction $\boldsymbol{e}_1$, the motion rule is

$$\Delta p_1 = \max(d_1 - T_1/k, 0),$$

where $T_1 = \mu k |d_0|$, i.e., a friction force threshold with friction coefficient μ. This motion rule is used to avoid slippery surfaces. The parameters k, μ, and T_0 can be controlled through the user interface. Figure 7 illustrates the idea behind the haptic rendering.



(a) Computation of $\boldsymbol{e}_0$ by finding the points on the tool surface that are in contact with the object.



(b) $\boldsymbol{e}_1$ is constructed by projecting $\boldsymbol{d} = (\boldsymbol{x} - \boldsymbol{p})$ onto the plane defined by $\boldsymbol{e}_0$.

Figure 7: Idea behind the haptic rendering.

5.3 Editing operations

Editing of the volume **S** is performed with the spherical tool described in Section 5.2. The tool can be either active or inactive. When the tool is active, all object voxels in **S** located within the tool boundaries will be affected by the currently selected editing operation. So far, we have implemented four basic editing operations: draw, erase, erode, and dilate. Erosion and dilation are binary morphology operators [Gonzalez and Woods 2002, Chapter 9] that are used to peel off a voxel layer and to add a voxel layer, respectively. See Figure 8 for a simple erosion example. The editing operations that are provided with haptic feedback is erase, erode, and dilate. Haptic feedback for drawing can be turned on as an option and is based on the gradient of **V** for feeling object boundaries.

Drawing and erasing are simple operations that can work directly on **S**, while erosion and dilation need an input volume and an output

Figure 8: The smiley is constructed by erosion.

volume. Therefore, a temporary volume $\mathbf{S}'$ is used. $\mathbf{S}'$ is a copy of $\mathbf{S}$ that is not modified while the tool is active. When the tool is deactivated, $\mathbf{S}'$ is updated according to the current $\mathbf{S}$.

6 Application example

In a recently started project in co-operation with the Dept. of Radiology at Uppsala University Hospital, we develop interactive segmentation methods as a part of liver surgery planning. As an initial part of the project, we have developed a method for segmentation of the liver from MR images. The images are of size $256 \times 256 \times 100$ voxels.

First, we apply pre-processing filters to the original data set, i.e., edge-preserving smoothing followed by gradient magnitude extraction. The gradient magnitude is used to construct a speed function for input to a fast marching segmentation algorithm [Sethian 1999]. As the next step, we use our drawing tool to create an initial seed-region inside the liver. The fast marching algorithm then propagates this region towards the liver boundaries. The propagation is fast where the gradient magnitude is low and vice versa. When the algorithm has converged, we examine the result and, if necessary, perform manual editing.

A screenshot from the application is shown in Figure 1. The initial segmentation result contains several artifacts due to "leaking", i.e., the contrast between the liver and the surrounding tissues is low. After manual editing, most of the artifacts are removed.

7 Results

To test the surface renderer, we generated a test image by sampling a 3D Gaussian function on a $128 \times 128 \times 128$ grid. We loaded the image into our enviroment and thresholded it at different levels to produce triangle meshes consisting of 20,000–100,000 triangles. We used the erosion operation to edit the mesh with tool radii $r = 5$ and $r = 10$ voxels. The resulting average update rates are given in Table 1, where it can be seen that the time for triangle generation increases linearly with the number of triangles. As a consequence, the effect of different tool radii decreases as the objects become larger. In the application example (Section 6), the marching cubes surface of the segmented liver consisted of 95000 triangles and was edited at frame rates between 10 and 12 frames per second. Tool radii between 1 and 10 voxels was used, and the number of tool sample points was 340. The haptic update rate was kept constant at 1 kHz which is the rule of thumb for perceptually convincing haptic feedback.

Table 1: Average update rates when editing triangle meshes with an erosion tool having a radius of r voxels.

	Update rate (frames/s)	
#Triangles	$r = 5$	$r = 10$
20,000	52	40
30,000	39	25
40,000	30	22
50,000	25	19
60,000	21	17
70,000	20	18
80,000	17	15
90,000	11	10
100,000	9	8

8 Conclusions and future work

The surface renderer that we have developed can be used for interactive editing of segmented objects. The efficiency lies mainly in the surface tracking and the index caching strategies. However, we note that when complex objects are triangulated, the huge number of triangles considerably slows down the rendering. To overcome this problem we will investigate how a triangle decimation algorithm could be incorporated and how the re-rendering can be improved to update only parts of the triangle mesh.

Regarding the haptic editing tools we are encouraged by these initial results, so we will extend the volume editor with several editing operations, arbitrarily shaped editing tools, and more sophisticated haptic rendering. Further more, we will investigate how to facilitate the creation of seed-regions by using haptic feedback based on the original volume $\mathbf{V}$. Ideally, the haptic feedback would force the user to draw inside the object, but using only gradient information for this purpose is not enough since it is easy to lose track of object boundaries when the contrast is low.

The segmentation method has shown promising results and we will continue development of the method. Evaluation of the segmentation method and the usefulness of the haptic editing tools will be conducted in coming work.

Acknowledgments

We would like to thank Doc. Ingela Nyström and Prof. Ewert Bengtsson at the Centre for Image Analysis for proofreading and useful comments. Prof. Håkan Ahlström and Dr Hans Frimmel at the Dept. of Radiology at Uppsala University Hospital are acknowledged for providing the MRI data. This work was funded by the Swedish Research Council, no. 2002-5645.

References

AGMUND, J. 2004. *Real-time surface rendering for interactive volume image segmentation in a haptic environment*. Master's

thesis, Uppsala University, Centre for Image Analysis. UPTEC F04 071.

AVILA, R. S., AND SOBIERAJSKI, L. M. 1996. A haptic interaction method for volume visualization. In *Proceedings of IEEE Visualization'96*, 197–204.

BÆRENTZEN, J. A. 1998. Octree-based volume sculpting. In *Proceedings of Late Breaking Hot Topics. IEEE Visualization'98*, 9–12.

GALYEAN, T. A., AND HUGHES, J. F. 1991. Sculpting: An interactive volumetric modeling technique. In *Proceedings of ACM SIGGRAPH'91*, 267–274.

GONZALEZ, R. C., AND WOODS, R. E. 2002. *Digital image processing*, second ed. Prentice Hall, Inc.

HARDERS, M., AND SZÉKELY, G. 2002. Improving medical segmentation with haptic interaction. In *Proceedings of IEEE Virtual Reality (VR'02)*, IEEE CS Press, IEEE Computer Society, 243–250.

IKITS, M., BREDERSON, J. D., HANSEN, C. D., AND JOHNSON, C. R. 2003. A constraint-based technique for haptic volume exploration. In *Proceedings of IEEE Visualization'03*, 263–269.

KANG, Y., ENGELKE, K., AND KALENDER, W. A. 2004. Interactive 3D editing tools for image segmentation. *Medical Image Analysis 8*, 1, 35–46.

KIM, L., SUKHATME, G. S., AND DESBRUN, M. 2004. A haptic-rendering technique based on hybrid surface representation. *IEEE Computer Graphics and Applications 24*, 2, 66–75.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface reconstruction algorithm. *Computer Graphics 21*, 4 (July), 163–169.

LUNDIN, K., YNNERMAN, A., AND GUDMUNDSSON, B. 2002. Proxy-based haptic feedback from volumetric density data. In *Proceedings of Eurohaptics 2002*, 104–109.

MONTANI, C., SCATENI, R., AND SCOPIGNO, R. 1994. Discretized marching cubes. In *Proceedings of IEEE Visualization'94*, IEEE Computer Society Press, Washington D.C., USA, R. D. Bergeron and A. E. Kaufman, Eds., IEEE Computer Society, 281–287.

PETERSIK, A., PFLESSER, B., TIEDE, U., HOEHNE, K. H., AND LEUWER, R. 2003. Realistic haptic interaction in volume sculpting for surgery simulation. In *Proceedings of IS4TM'03*, Springer-Verlag Berlin Heidelberg, N. Ayache and H. Delingette, Eds., no. 2673 in LNCS, 194–202.

RUSPINI, D. C., KOLAROV, K., AND KHATIB, O. 1997. The haptic display of complex graphical environments. In *Proceedings of ACM SIGGRAPH'97*, ACM SIGGRAPH, 345–352.

SETHIAN, J. A. 1999. *Level set methods and fast marching methods*. Cambridge University Press.

SHEKHAR, R., FAYYAD, E., YAGEL, R., AND CORNHILL, J. F. 1996. Octree-based decimation of marching cubes surfaces. In *Proceedings of IEEE Visualization'96*, 335–ff.

THURFJELL, L., MCLAUGHLIN, J., MATTSSON, J., AND LAMMERTSE, P. 2002. Haptic interaction with virtual objects: The technology and some applications. *Industrial Robot 29*, 3, 210–215.

VIDHOLM, E., TIZON, X., NYSTRÖM, I., AND BENGTSSON, E. 2004. Haptic guided seeding of MRA images for semi-automatic segmentation. In *Proceedings of IEEE International Symposium on Biomedical Imaging (ISBI'04)*, 278–281.

WILHELMS, J., AND VAN GELDER, A. 1992. Octrees for faster isosurface generation. *ACM Transactions on Graphics 11*, 3, 201–227.

WYVILL, B., MCPHEETERS, C., AND WYVILL, G. 1986. Data structure for soft objects. *The Visual Computer 2*, 4, 227–234.

Collaborative 3D Visualizations of Geo-Spatial Information for Command and Control

Lars Winkler Pettersson*
Department of Information Technology
Uppsala University

Ulrik Spak†
Swedish National Defence College

Stefan Seipel‡
Department of Information Technology
Uppsala University
Department of Mathematics and Computer Science
University of Gävle

Abstract

We present a prototype command and control system that is based on view-dependent co-located visualizations of geographically related data. It runs on a 3D display environment, in which several users can interact with view consistent visualizations of information. The display system projects four independent stereoscopic image pairs at full resolution upon a custom designed optical screen. It uses head tracking for up to four individual observers to generate distortion free imagery that is rendered on a PC based rendering cluster. We describe the technical platform and system configuration and introduce our unified software architecture that allows integrating multiple rendering processes with head tracking for multiple viewers. We then present results of our current visualization application in the field of military command and control. The command and control system renders view consistent geographical information in a stereoscopic 3D view whereby command and control symbols are presented in a viewpoint adapted way. We summarize our experiences with this new environment and discuss technical soundness and performance.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics

Keywords: Distributed Rendering, Networked Virtual Environment, Display Environments, Stereoscopic Projection, GIS

1 Introduction

The demand for visualization in command and control systems follows an increased focus on Network-Centric Warfare [Sundin and Friman 2000]. Information from knowing exact positions and status of individual military objects such as troops, tanks, airplanes, etc. will only give an advantage if the information can be visualized in an efficient and useful way in an environment suitable for collaboration. In our work towards increasing the amount of directly available information to a user, we have chosen to focus on stereoscopic visualization. The large amount of military information related to spatial geographic data makes command and control systems suitable for stereoscopic visualization. We suggest that to achieve efficient stereoscopic visualization in a collaborative environment each user's perspective of the display must be view-dependent with the visualized model co-located in the same presentation volume. The first characteristic, view-dependent perception is important for enhancing the visualization by presenting each user in the display environment with textual tags and symbols oriented towards their gaze. The second characteristic, the visualization of a co-located model, ensure that all the users in the environment perceives the model as one and can interact with it. To describe the visualization technique based on these characteristics we use the term view-dependent co-located visualization.

Computer generated stereoscopic visualization has a vast history. Only a decade after the first computers were brought into existence, volumetric displays were invented [Hirsch 1961; Ketchpel 1964] and a few years after the head mounted display (HMD) [Sutherland 1968]. The stereoscopic visualization techniques developed since then are to different degrees suitable for view-dependent co-located visualization.

Autostereoscopic displays do not require the viewer to use a head worn filter, since the display itself generates a stereoscopic visualization. The two most common classes of autostereoscopic displays for displaying general three-dimensional information are volumetric displays and parallax displays [Halle 1997]. Volumetric displays can have an unlimited number of viewers due to their nature of sweeping out a volume in space, but they do not allow for view-dependent visualization since each point in visualization space can be seen by several users. Parallax displays such as computer generated holograms and lenticular array displays makes view-dependent co-located visualization theoretically possible but practically hard to achieve. In the case with computer generated holograms the data rate needed to produce a stereoscopic visualization is very high and still not practically viable. Lenticular array displays uses an optical layer that for each additional viewer reduces the resolution of the image. For more than two face to face collaborating users this technology will not suffice. HMDs used for displaying stereoscopic visualizations can either represent the surrounding environment through a virtual setup, a video see-through setup or an optical see-through setup [Azuma 1997]. Command and control work is highly dependent on face to face collaboration. HMDs representing the surrounding environment by a virtual or video captured stream will hinder the detection of subtle facial expressions used in face to face collaboration. Optical see-through HMDs could be an efficient alternative for collaborative work, but the technique is sensitive to incorrect head tracking with the requirement of registering six degrees of freedom (6DOF) at below 10 ms latency [Azuma 1997]. Compared to HMDs, stereoscopic screen visualization with

*e-mail:lwp@it.uu.se
†e-mail:ulrik.spak@fhs.mil.se
‡e-mail:ssl@hig.se

shutter glasses or passive polarizing glasses can be achieved efficiently by means of three degrees of freedom tracking of the viewpoint. It is therefore less affected by angular tracking errors and net latency in the registration equipment. Stereoscopic presentation techniques based on active shutters generate frame rate dependent flickering when two or more users share a presentation area in a temporal multiplexing scheme [Agrawala et al. 1997]. In order to avoid flickering when shutter glasses are used for more than one independent observer, the display can be spatially divided into separate viewing zones for each user [Kitamura et al. 2001]. This approach works at the cost of significantly reduced effective resolution and spatial viewing area of the display. A recent paper [Hedley et al. 2002] describes a technique for presentation of geographic information in a multiple-viewer collaborative setting. In their work, view-dependent co-located visualization is accomplished by using opaque head mounted displays for each independent observer. The geographic information is presented in the environment using a video see-through augmented reality (AR) system. The system correlates geometries in a video based stream to fiducials where one layer is used to present the geographic information and others as lenses looking onto the geographic information. Another approach to 3D visualization of geospatial information is VGIS [Lindstrom et al. 1997]. Their work focuses on data handling and level of detail algorithms and not on a multiple viewer collaborative setting. Data is preprocessed and presented using a bottom up technique which acquire and presents data in three stages: Preprocessing offline, Intermediate online processing and Real-time online processing. Their system interacts with simulations using Distributed Interactive Simulations (DIS) and can in the preprocess stage access data from Arc/INFO GIS servers.

2 The Multiple Viewer Display Environment

In our research we use a novel display environment that is composed of several hardware and software components. At the very bottom we find a display which presents simultaneously the content of eight independent image sources. These images are presented within the same physical area on-screen and can therefore be perceived in the same locus by different viewers. The eight co-located images can be used to either provide four independent viewers with individual stereoscopic imagery or to serve eight independent observers with monocular views.

2.1 Display and graphics hardware

The physical display system is essentially an integrated retro-projection display system, in which the projection screen is oriented horizontally. This approach provides a viewing metaphor virtual world that is perceived as a bird's eye view. Similar viewing configurations have been presented earlier for instance in medical application scenarios. What is novel with the display environment presented here is the fact that eight independent digital image projectors are used to rear-project images from eight independent image sources upon the same physical screen area. Each two projectors are projecting from one of the four main directions upon the screen. The square screen has a size of 0.8 by 0.8 meters. Projection images are horizontally adjusted such that the digital image centers align with the centre of the square screen. The visible pixel resolution of each projection image on-screen is 768 by 768 pixels. All eight projection-images are superimposed upon the same physical screen area, but pairs of two are rotated 90 degrees or 180 degrees, respectively in relation to one another. The separation of

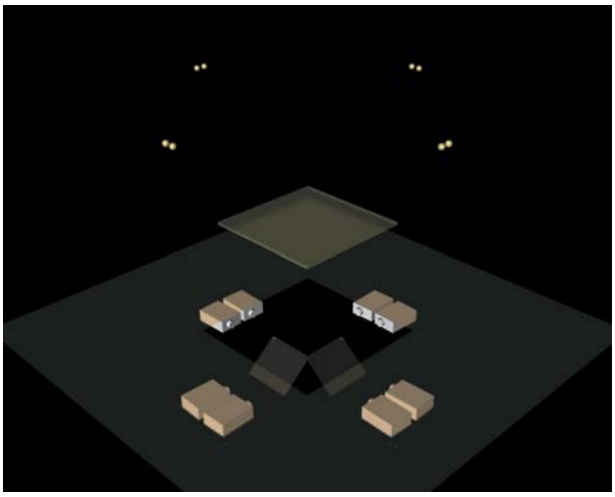


Figure 1: Configuration of the multiple viewer co-located retro projection display.

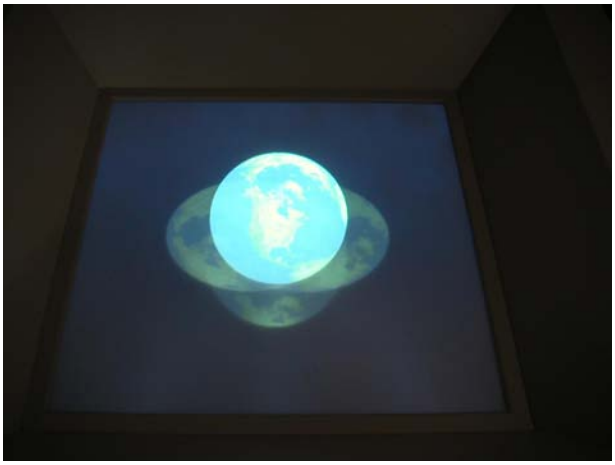


Figure 2: View upon the co-located display as seen by one eye of the observer. The picture shows the viewpoint corrected monocular view as well as (subdued) the pictures aimed for the remaining other viewers.

stereo image pairs is accomplished by using polarizing filters in front of the image projectors rather than using an active shutter system with temporal multiplexing. Figure 1 shows schematically the configuration of the display system and the viewer's eye positions in relation to the horizontal screen. One component of the display that is not further described in this place is a special optical design for the retro-projection screen, which allows the projection images to penetrate the screen primarily along the direction of the optical axis without loosing polarization. In consequence, a viewer being located at one side of the screen perceives only two of the eight projection images and, when using passive stereo glasses, perceives only one of the two images on either eye. Figure 2 shows the viewpoint adapted projection image for an observer looking from one side of the horizontal display. In this picture, also the view of the remaining three observers upon the same object is evident. The picture is taken without polarizing filters, and for each observer only a monocular view has been rendered. This display system is driven by a small cluster of four commercial off-the-shelf computers that are interconnected with a Giga-Ethernet local area network. Each of

the four computers is equipped with a NVidia QuadroFX graphics cards and renders two pipes. Hence, for a four-person configuration, each computer renders the stereo image pair for one observer. For the purpose of head position tracking, a magnetic tracking system is used (Ascension Flock of Birds) that is connected to a separate computer.

2.2 A model for distributed rendering

As is evident from the technical description of the display environment, the rendering process must be delegated upon different rendering nodes, which requires some means of synchronization of both the scene updates and the viewer dependent rendering parameters (e.g. head position and viewport orientation). Distributed rendering and networked VR systems have been described earlier and in various places in literature. These approaches can be divided into two classes: The first class are systems that support rendering into tiled displays (e.g. for very large projection walls). The purpose of these kind of rendering architectures is to distribute graphic states and primitives at a very low level, whereby typically the nodes in a rendering cluster render a sub-portion of one view upon the same scene. A well-known system is the Chromium architecture [Humphreys et al. 2002]. Since these systems work at the very low end of the graphics rendering pipeline, they cannot be used in applications that require node specific 3D scene content. The second class of networked based rendering systems tackles the problem at the high level graphical API. Commonly found in scene-graph programming tools, they provide fully transparent propagation of changes in the scene objects and scene graph. Early typical examples are the DIVE system [Carlsson and Hagsand 1993] or more recently Net Juggler [Allard et al. 2002]. The comfort of having a transparently shared and consistent scene database distributed over a network is achieved at the cost of performance. Still, in most networked VR environments, where the rendering results of various nodes are viewed at different physical locations, certain delays in scene updates can be tolerated. However, for physically co-located view-ports as in tiled displays or as in our case of superimposed displays these distribution models are not feasible, because delay becomes a strongly distracting visual artifact. Typical for our application scenario is an unusual blend of shared data and node-specific states that affect rendering in the local node. The combination of all following requirements renders our application different from most other networked or cluster based rendering applications:

- Large and complex parts of the scenario (e.g. terrain) are static and do not require network propagation
- Only relative few shared objects in the scene do change/move and need to be distributed among rendering nodes
- Some parts of the scene (observer dependent symbol orientation) are node specific content and pertain to the local node only
- View-port orientation and off-axis projection parameters are node/observer specific and change continuously

Due to these particular viewing conditions, none of the existing networked rendering architectures show to be really efficient solutions for these purposes. Systems like Chromium do not cope well with node specific scene content since they forward the graphical primitives from one host application to several rendering nodes. Here the nodes render view-port selectively rather than scene specific content. The classical network based VR systems on the other hand introduce too much overhead considering, in our application, the very few required scene updates that must be maintained almost instantly at all nodes. In order to solve this problem independently from the underlying scene-graph toolkit used for application development, we developed a very slim distribution model for fast exchange of data that must be shared among rendering nodes and among tracking devices connected to the environment. At the core of this distribution model we implemented a virtual shared memory model that allows for applications to allocate and subscribe to arbitrary memory partitions. The shared data repository resides on a server and uses either TCP/IP or UDP services for propagation of state changes between clients and the shared repository. Applications can enforce strict data consistency, when utilizing the built-in locking mechanisms of our shared memory API. However, strict consistency is not always needed and can be sacrificed in favor of improved overall network performance. One typical such example is frequent state changes of animated 3D objects as a result of direct user manipulation. In the practical situation it is more favorable to have a low latency motion of that kind of objects that is visible simultaneously for all viewers even though some of the states along the motion might be lost. Based on this virtual shared memory, our distribution model implements so-called data pools, where information is shared among the clients, that is relevant for distributed rendering and eventually for communication with other applications. Our distribution model encapsulates mainly three types of objects into the shared data pool. They are:

Projectors A projector entity contains configuration parameters for a specific viewing frustum to be rendered by one or several rendering nodes in the cluster. This comprises projection frustum parameters in physical real world coordinates as well rendering pipe parameters (viewports) and references to sensors, to which the viewing position is locked.

Sensors The sensor entity is an abstract object that pertains to the values received by some specific input device. It keeps a description of the semantics of the data delivered by the input device as well as the actual data values measured with the device. A sensor can for instance be a representation of any physical tracking device, but it could also be a logical device that integrates data from different physical input devices

Messages Messages entities are used to communicate messages between processes. Message passing is accomplished by subscribing to message entities. Accessing a message entity broadcast the content of that message entity to all subscribing clients.

More technical details on this distribution model can be found in [Lindkvist 2002]. An experimental performance study of this distribution model has been presented by [Seipel and Ahrenberg 2002]. In the following section we describe, how this distribution model is used in order to integrate multiple rendering processes into one visualization application. Figure 3 illustrates the relations between the different processes involved into the visualization system.

Four *rendering processes* that run concurrently on either network node generate the visual output in the horizontal display. These rendering processes are identical processes and therefore provide the same functionality in regard to scene behavior and user interaction. At startup these render processes initialize their local scene graphs based on a common shared scene database. At this stage, the only difference among the four rendering applications is the parameterization of the off-axis projections, their viewports on-screen and the associated sensor entity for head tracking. This process specific configuration is loaded from a local configuration file that keeps a reference to the actual sensor and projector entities in the shared repository containing the relevant data. The rendering applications do not maintain the actual data by themselves. Instead, sensor and projector data is read only by the rendering processes to update the display appropriately. The actual manipulation of the current

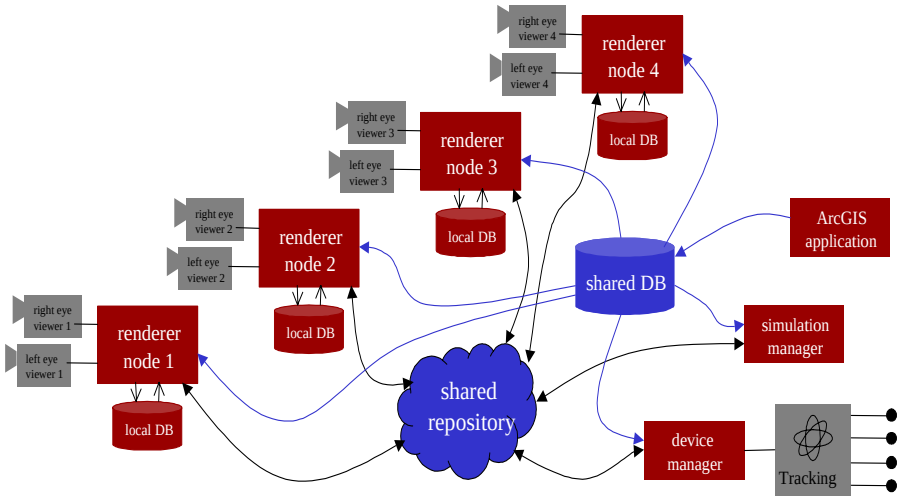


Figure 3: The drawing shows schematically how input and output device (gray), processes (red) and shared resources are communicating to flexibly synchronize the distributed rendering process in the co-located visualization environment.

configuration data for different projectors as well as the update of current tracking data in the shared repository is performed by separate processes. Dedicated *device managers* are running as independent processes in the network and continuously translate tracking values from devices into sensor values shared in the repository. In this process, hardware specific device parameters are abstracted to logical sensor devices that have certain agreed upon properties (e.g. representing a 3D position, or a set of three vectors). For the proper operation of the entire visualization environment, it does not matter what number and actual type of input devices are used to capture the values of a sensor entity. In fact, a device manager process could read measurements from different tracking systems and combine them into a logical sensor entity with higher accuracy. Or it could combine several positional device values into on position/orientation sensor. Also, switching or replacing input device hardware during operation of the visualization is only a matter of starting up a new physical device with its corresponding device manager process. The principle coordination of all visualizations shown in this environment is carried out by a server process, which in this context is called a *simulation manager*. The simulation manager is the administrator of the shared repository and main negotiator between the nodes. It continuously updates almost all entities in the shared repository and it routes messages between the different processes. One of the main tasks of the simulation manager is to update projector entities as a consequence of updated sensor values due to head motion of individual users. Since projector entities are transparently shared through the repository, all applications that have subscribed to a currently updated projector entity will autonomously update their output on screen. In that sense updates of the viewpoints the individual rendering clients can be controlled remotely and simultaneously by other processes. Changes of projector entities (hence client viewing parameters) could also be triggered by other external events. One relevant example is the change of the geographic context and cartographic layers within the 3D visualization. Using commercially available *geographical information systems* (GIS), an operator can chose the desired geographic region and map content and export it to a predefined server partition. The simulation manager monitors continuously all recent updates in that partition and broadcasts messages to the rendering applications to initiate reloading or updating of their local scene data (maps). This is a rather simple and straightforward method to control the flow of geographic visualizations in the environment described here. It proves efficient because it does not require reprogramming of external applications and it uses established GIS tools to retrieve the geo-spatial data. Other tasks performed by the simulation manager are controlling of workflow within the 3D visualization by means of other user interfaces, or to interface with other external programs, which in our case, simulate military scenarios. In all cases of simulations, the simulation manager acts as a computing process and event dispatcher only. The actual visual simulation and animation of graphical objects is handled identically by the rendering clients, which administrate their local scene graphs. Therefore, animated graphical processes in the distributed visualization follow a dead-reckoning approach [Singhal and Zyda 1999], whereby animation of objects is performed autonomously at the local clients without synchronization between the animation steps.

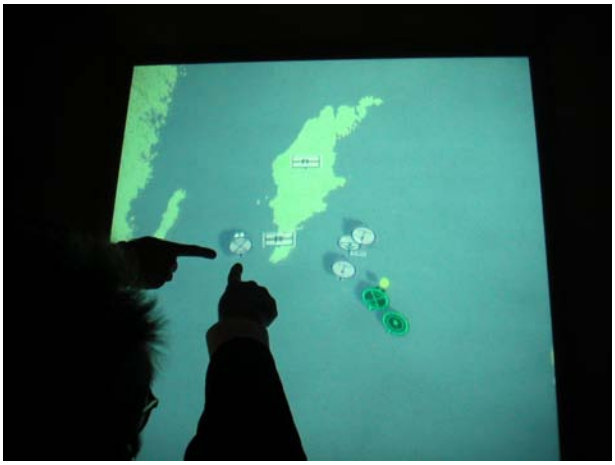
3 View Dependent Content Visualization in Geo-Spatial Context

Visualizations in command and control systems present geo-spatial data as well as content visualization where symbols describing military functions and objects have a prominent role. The described multiple viewer display environment is capable of presenting military entities using 3D objects but research suggests that classification and identification of military entities using 3D objects in most cases are more difficult and takes longer time to perceive than using traditional symbols [Smallman et al. 2001]. We have chosen to use traditional military domain 2D symbols in a 3D environmental setting. Since the multiple viewer display environment is capable of view-dependent co-located visualization, the traditional 2D symbols can be presented independently for each user of the display environment. A trivial but effective view-dependent rendering technique has been used to separate the presentation of military domain symbols from the presentation of geographic informa-

Figure 4: The control program for military domain visualization

tion. The view-dependent rendering technique presents the military domain symbols oriented orthogonally to the gaze of each viewer. This presentation technique where an object is correctly rotated and orthogonal to the viewers gaze is called frontoparallel presentation. Frontoparallel presentation should as well be effective for presenting text labels and other geographically tied symbols.

In figure 5, two users are shown collaborating in the multiple viewer display environment. They are provided with two perspectives of the same geographic area. For each view the symbols are frontoparallel towards the head tracked position of the user. The two views are the view from south, figure 5(a) and the view from the west, figure 5(b). In this scenario the users are discussing one particular symbol, and as can be seen in the figure, the symbol they point to is frontoparallel to both users views while remaining in the same position in the geographic area. The observable difference between frontoparallel and flat horizontal presentation is used to represent inactive objects, which are rendered flat in the horizontal plane but still oriented towards the spectator.



(a) South

(b) West

Figure 5: Two users discuss a situation in the same geographic area with military domain symbols frontoparallel to each users perspective.

4 Discussion and Conclusion

We have used our distribution model based on a shared repository to develop different visualization scenarios for the multiple display visualization environment. From a perceptional point of view, we have encountered that latencies in our distributed rendering process are at a very short time levels, which results in no perceived delays of animated structures among different viewers. A supporting factor in this context is that individual observers in the viewing environment do not actually perceive the views of the other collaborators. Still, the users must be able to get a common sense in regard to the temporal alignment when objects have started moving in the scenario and when they have terminated their motion. This temporal alignment is the common sensation of a shared dynamic scene is however, not as time critical and does not require strict frame-to-frame synchronization of the shared scene graph. Given our current configuration, whereby one rendering node renders the two stereoscopic views for one observer, the rendering of the left-eye and right-eye frames are always strictly frame synchronized, since they are rendered into different viewports within the same physical frame buffer. Hence, undesired parallax artifacts due to potential frame delays on the left eye and right eye, respectively, are not possible. In fact, the distribution model described in this paper would allow for quickly scaling up the rendering process upon 8 independent rendering nodes. This would not even require a recompilation of the system. Instead eight rendering processes would be instantiated on eight network nodes, whereby the configuration file for the local rendering processes would be modified to specify which respective projector and sensor entity should be used. In this configuration, there is a potential risk for frame delays between the observer's left and right eye view. Since our distribution model advocates a dead reckoning approach to distributed animation, asynchronous frames due to different rendering performances of the local nodes are likely. Since this type of configuration is only hypothetical, we have only conducted informal tests on this setting, which showed, that perceivable artifacts become visible. However, our shared repository approach is general enough such as to be used for synchronization of objects animations, even if it would not guarantee strict concurrency. Our measurements from previous experiments showed [Seipel and Ahrenberg 2002], that the

number of frame out-of-synch can be kept to as low as two frames in a sequence of 100 animation frames for a modestly sized scene. One main objective of our shared repository approach was to enable and maintain flexible configurations of viewports and viewer positions. The shared memory model is based on small data packets that are distributed asynchronously in order to minimize low latencies in network propagation. For the viewpoint-dependent visualization this is an important criterion to maintain the illusion of virtual 3D objects. The tracking values supplied by various device managers in our environment, are propagated rapidly to the local rendering nodes in order to accomplish dynamic viewing conditions. Our preliminary user studies indicate that delays in viewing frustum updates using sensor objects are not perceivably longer as compared to configurations that interface directly to the tracking equipment. We measure this in terms of the user's subjective experience of the three dimensional appearance of objects, which is largely dependent on dynamic parallax. In this context we can state, that three-dimensional appearance of virtual objects is equal for configurations that use shared sensor-entities and for configurations that interface immediately to the tracking hardware. Our visualization system has been developed with the goal to support flexible viewport handling and complex viewing parameterizations. This was a property, which we have missed in other previously published distributed rendering architectures (see section two). The solution that we have described here allows for controlling the visualization from other applications than the actual rendering processes. In our current visualization application, different phases of a military scenario are controlled with a pen-based wireless computer interface as shown in figure 4. In fact, a separate process performs the selection of the appropriate viewport and perspective to be rendered by a rendering client at a given time. Rendering clients are naïve processes in regard to viewpoint control. This opens up for new opportunities in collaborative work. The scenario presented above demonstrates collaborative work in the same geographical context. Another potential working situation would be that a workgroup coordinator (from a separate user interface), delegates planning tasks in different topographic regions to the four different users at a time. In this situation, the views upon the geographical visualization would temporarily be split up into distinct local regions, and after task completion, all observers would again gather around the same global view upon the scenario. This feature as well as user triggered private detail views into the geographic context can be managed very flexibly by means of the projector-entities in the shared repository. Another example that demonstrates the flexibility of the distribution model is the case of a user walking around the horizontal display. While walking from one side over to the next side of the screen, the user will, due to the optical properties of the screen, leave one viewing zone and enter the next one. Different rendering nodes will provide the views within the respective viewing zones, and their viewport orientation will shift 90 degrees. In the military domain visualizations it has been assumed that frontoparallel 2D symbols are easier to perceive than 2D symbols presented in the plane of a horizontal display. An experiment comparing the efficiency in identifying frontoparallel and flat horizontal presentation has been carried out and gathered data are being evaluated. Preliminary results indicate that frontoparallel presentation indeed has a lower error rate and faster recognitions times when compared flat horizontal presentation.

5 Future Work

The visualization environment as described in this paper suffers from apparent crosstalk between stereo images pairs, as is the case for most visualization systems that present different information for multiple viewpoints in the same physical display area. However, according to interview studies with domain experts using the visualizations, stereoscopic crosstalk was not reported as major disturbing artifact. In order to quantify this further it would be interesting to study in more detail how tolerant the human visual system is in regard to visual crosstalk between the stereo image pairs and other visual noise. These studies would require a definition and setup of a measuring procedure that is generalized and portable to other display environments. In regard to the frontoparallel presentation of 2D symbols, we are conducting further experimental research on the human's capability in reading this form of presentation as compared to traditional flat presentations.

6 Acknowledgements

The work presented in this paper is result of a sub-project funded by Swedish National Defence College (SNDC) in the framework of project AQUA. Technical details on the display hardware used are proprietary of S. Anders Christenson at SNDC and Peter Segerhammar at VAB and are published elsewhere by their authors. We acknowledge their admission to reproduce schematically the working principle of the multiple projector display device as required for the comprehension of the presented multiple viewer 3D visualization system.

References

AGRAWALA, M., BEERS, A. C., MCDOWALL, I., FRÖHLICH, B., BOLAS, M., AND HANRAHAN, P. 1997. The two-user responsive workbench: support for collaboration through individual views of a shared space. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniquesarrraraa*, ACM Press/Addison-Wesley Publishing Co., 327–332.

ALLARD, J., GOURANTON, V., LECOINTRE, L., MELIN, E., AND RAFFIN, B. 2002. Net juggler: Running vr juggler with multiple displays on a commodity component cluster. In *Proceedings of the IEEE Virtual Reality Conference 2002*, IEEE Computer Society, 273.

AZUMA, R. 1997. A survey of augmented reality. *Presence, Teleoperators and Virtual Environments 6*, 4, 355–385.

CARLSSON, C., AND HAGSAND, O. 1993. Dive - a platform for multi-user virtual environments. *Computers and Graphics 17*, 6, 663–669.

HALLE, M. 1997. Autostereoscopic displays and computer graphics. *Computer Graphics 31*, 2, 58–62.

HEDLEY, N. R., BILLINGHURST, M., POSTNER, L., MAY, R., AND KATO, H. 2002. Explorations in the use of augmented reality for geographic visualization. *Presence: Teleoper. Virtual Environ. 11*, 2, 119–133.

HIRSCH, M., 1961. Three dimensional display apparatus. U. S. Patent No. 2,967,905.

HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P. D., AND KLOSOWSKI, J. T. 2002. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 693–702.

KETCHPEL, R. D., 1964. Three-dimensional cathode ray tube. U. S. Patent No. 3,140,415, July.

KITAMURA, Y., KONISHI, T., YAMAMOTO, S., AND KISHINO, F. 2001. Interactive stereoscopic display for three or more users. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 231–240.

LINDKVIST, M. 2002. *A State Sharing Toolkit for Interactive Applications*. Master's thesis, Royal Institute of Technology.

LINDSTROM, P., KOLLER, D., RIBARSKY, W., HODGES, L., AND FAUST, N. 1997. An integrated global gis and visual simulation system. Misc GIT-GVU-97-07, Georgia Institute of Technology.

SEIPEL, S., AND AHRENBERG, L. 2002. Distributed rendering in heterogeneous display environments - a functional framework design and performance assessment. In *Proc. Annual SIGRAD Conference 2002*.

SINGHAL, S., AND ZYDA, M. 1999. *Networked virtual environments: design and implementation*. ACM Press/Addison-Wesley Publishing Co.

SMALLMAN, H. S., OONK, H. M., AND JOHN, M. S. 2001. Symbicons: Advanced symbology for two-dimensional and three-dimensional displays. Tech. Rep. TR-1850, SPAWAR Systems Center San Diego.

SUNDIN, C., AND FRIMAN, H., Eds. 2000. *ROLF 2010 - The Way Ahead and The First Step*. Elanders Gotab.

SUTHERLAND, I. E. 1968. A head-mounted three-dimensional display. In *AFIPS Conference Proceedings*, Thompson Book Co., vol. 33, 757–764.

Work in Progress: Context Aware Maps

Anders Henrysson*
Norrköping Visualization and Interaction Studio
Linköping University, Norrköping, Sweden

Mark Ollila†
Norrköping Visualization and Interaction Studio
Linköping University, Norrköping, Sweden

Abstract

We look at how analog and digital maps can be augmented with context related information. In the analog case we use Augmented Reality to superimpose virtual objects onto the view of a printed map. For user friendliness we use standard mobile phone for Augmented Reality. To do this we have ported the ARToolkit to Symbian. We have implemented two case studies where printed maps are augmented with traffic-data by attaching markers to them.

1 Introduction

The importance and usage of printed map through history needs no further introduction. Maps can be found almost everywhere to help us orient in our environment. In their analog e.g. printed form they provide us static geospatial information. In recent years we have also seen the introduction of digital maps in applications such as car navigation, services on the web that let you search for a particular address and so on. Digital maps are more dynamic but also more expensive to employ and are overwhelmingly outnumbered by analog ones.

Ordinary 2D maps can be extended into several dimensions using Augmented Reality (AR) as shown in [Bobrich and Otto 2002]. AR is about enhancing a users view of the real world by projecting computer generated graphical information onto it. Another way to see it is that the visualization domain is projected onto the problem domain. To do this the position and orientation of the uses head must be tracked by the system. Tracking can be done with ultra sound, GPS, magnetic sensors, video etc. We are working with the ARToolkit [ARToolkit], which is based on video tracking of markers. A marker consists of a square with a pattern. The pattern is known and is used to identify the marker while the square is used for calculating the camera position and orientation relative to the reference coordinate system centered at the marker. These extrinsic parameters are used as the modelview matrix in a 3D graphics pipeline while the intrinsic parameters obtained during calibration of the camera are used for the projection matrix. The overlaid graphics can thus be rendered with the correct perspective.

In our research we are looking at Ubiquitous Mobile Augmented Reality [Henrysson and Ollila 2004]. We want the platform to be mobile and ubiquitous. We have chosen to work with Smartphones, which have the necessary capabilities for AR i.e. camera, CPU, storage and display. We have ported the ARToolkit to Symbian, which is one of the dominating Smartphone platforms. The purpose of our research is to visualize context related information where AR is one of the visualization modes. The idea is to retrieve information based on personalization and context awareness data and present this information using appropriate visualization mode considering the spatial relationship between information and environment.

*e-mail: andhe@itn.liu.se
†e-mail: marol@itn.liu.se

2 Context Aware Maps

The idea behind context aware maps is to augment analog and digital maps with context related information. Context Awareness is an area where sensor data such as position, time, temperature etc. is used to optimize configurations and services. We have focused on time and position. To augment the maps, markers with known sizes and patterns have to be added. We must know the scale of the map and the position of the markers for the superimposed graphics to have the correct scale and position.

3 Implementation

We have implemented two case studies; a tram station map augmented with an animation showing the current tram positions and a map of the Nordic region augmented with air traffic data. The hardware setup consists of a Nokia 6600 running Symbian and Series 60. It has a 0.3 megapixel camera with a QQVGA viewfinder resolution and 106 MHz ARM9 CPU.

The tram stop application is driven by timetable data and shows the current positions of the trams trafficking the area shown on the map. Since this map was originally in bitmap format the implementation was straightforward. First we obtained the positions of the tram stops in pixels relative to the composited marker and then we translated the distance in pixel space to distance in coordinate space. The rendering is done in 2D with sprites representing the trams (see Figure 1).

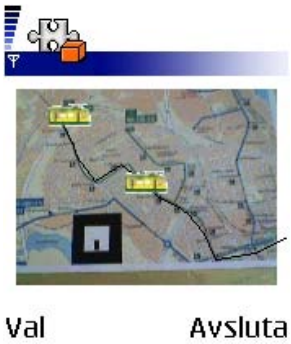


Figure 1: Map with marker and screenshot from mobile device.

The flight data application uses authentic flight data from the Air Traffic Control (ATC) [Lange et al. 2003] project at NVIS. In ATC flight trajectories are visualized together with weather information. The trajectories can be manipulated interactively to avoid collisions and bad weather conditions. As a first step we have augmented a

printed map of the Nordic region with a 3D flight trajectory between Arlanda and Kastrup (see Figure 2). The plane is represented with a simple polygon to reduce complexity since we are using pure OpenGL ES. The flight information is given in longitude and latitude, which are also the coordinates of the map. We have approximated with Cartesian coordinated to avoid spherical to planar mapping.

Figure 2: Flight data

We have so far only used one marker, which limits our field of view, but more markers can be used to enlarge the trackable area.

4 Discussion and Future Work

We have shown how to tie context-related computer generated information to printed maps using AR on such a ubiquitous device as a Smartphone. This can be applied to any printed map provided its coordinate system and the relative positions of the markers are known. A drawback of our current solution is that markers have to be known in advance, but there exist solutions [Spotcode] where the pattern consists of a sequence similar to a bar code that can be translated to a number and used to fetch information from a server. Such networking solutions are essential for context aware maps that require up-to-date information.

The current plan for future work is to speed up the software using fixed-point math. After that we will look at how to extend the tracking to be more independent of markers. One technique that will be useful for augmented maps is texture tracking [Kato et al. 2003] where texture features can be used to track the camera. Other methods that will be studied are GPS, optical flow and inertia sensors.

We would also like to study the interaction capabilities of the Smartphone and compare it to using a wand in front of an immersive workbench, which is the case with the current ATC implementation.

References

ARTOOLKIT. www.hitl.washington.edu/artoolkit/.

BOBRICH, J., AND OTTO, S. 2002. Augmanted maps. In *Symposium on Geospatial Theory, Processing and Applications*, Ottawa, Canada.

HENRYSSON, A., AND OLLILA, M. 2004. Umar - ubiquitous mobile aygmanted reality. In *3rd International Conference on Mobile and Ubiquitous Multimedia*, Washington, USA.

KATO, H., TACHIBANA, K., BILLINGHURST, M., , AND GRAFE., M. 2003. A registration method based on texture tracking using artoolkit. In *2nd IEEE International Augmented Reality Toolkit Workshop*, Waseda Univ., Tokyo, Japan.

LANGE, M., HJALMARSSON, J., COOPER, M., YNNERMAN, A., AND DUONG, V. 2003. 3d visualization and 3d and voice interaction in air traffic management. In *SIGRAD2003, The Annual SIGRAD Conference.*, Ume, Sweden.

SPOTCODE. www.highenergymagic.com/spotcode/.

Work in Progress: GPU-assisted Surface Reconstruction and Motion Analysis from Range Scanner Data

Daniel Wesslén*
University of Gävle

Stefan Seipel†
University of Gävle

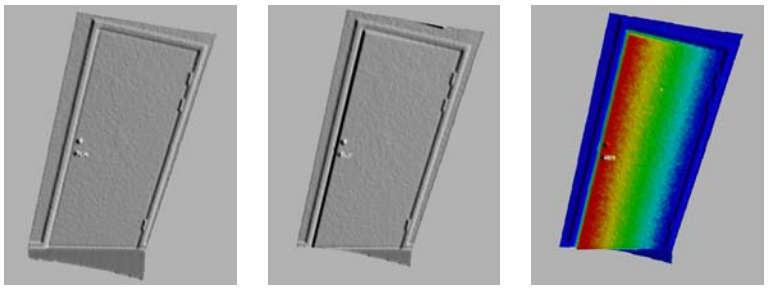


Figure 1: Reconstructed surfaces from two scans of the same door and exaggerated, color coded difference.

Abstract

We present a method for rapid GPU-assisted surface reconstruction from range scanner data producing meshes suitable for visualization and analysis of very slow-moving objects from multiple scans of the same area.

CR Categories: I.3.6 [Computer graphics]: Methodology and techniques—Graphics data structures and data types, Interaction techniques

Keywords: surface reconstruction, meshing, point clouds, laser scanner

1 Introduction

Usually, range scanners or other point measuring equipment is used to sample an object from which a surface will be generated, reduced or otherwise processed. The final model is expected to be used on many occasions and therefore the time spent on constructing it is considered expendable. [Amenta et al. 2002; Bajaj et al. 1995; Curless and Levoy 1996]

We are facing a different situation—scans are to be acquired at regular intervals and the difference between them used to detect motion. Analysis will be both manual and automatic. A display will be provided for users to look at the data, which will have to be exaggerated for differences to become detectable. Automatic analysis will run continuously, alerting a user when potentially interesting data is detected.

*e-mail: dwn@hig.se
†e-mail: ssl@hig.se

In this scenario meshes become irrelevant when new data is available, so time spent on creating a perfect mesh is wasted—especially if nobody is looking at the display, in which case it will be used only once for automated analysis.

It is also desirable to quickly be able to bring up a mesh representing changes between any two previous scans, providing further incentive to find a solution that executes as fast as possible.

1.1 Previous Research

Automated surface reconstruction from point cloud data sets and depth images has seen much research, though usually focused on finding meshes that optimally fit the input point set in some way or that use as few triangles as possible. We have found no existing method suited for our conditions.

2 Surface Reconstruction

Surfaces are generated as regular grids, much like the input data. Optionally, meshes may be generated from multiple point sets at once, in which case corresponding vertices in the two meshes will correspond to the same direction from the scanner. This is the most common mode of operation since it facilitates motion analysis.

Reconstruction is fairly strightforward—input data is cleaned up, matched to the target raster, cleaned up again, and normals are then generated—the interesting part is that this process is almost entirely performed on the graphics card.

2.1 Cleanup

Input point sets may contains both valid points, invalid points where no measurement was recorded, points where the light was reflected from multiple surfaces, and spurious incorrect values which do not correspond to any surface. When light is reflected from more than one surface the reported distance will lie between the correct alternatives, disconnected from both surfaces.

The first task is thus to clean the data by removing invalid points. Apart from actually loading the data, this is the only part of the algorithm that is currently performed by the CPU.

A simple sweep of the data is performed, replacing the values of all points which are invalid or disconnected from the surface. Values are considered disconnected if their distance from neighbouring values in all directions is greater than a threshold. Coordinates are also restored to scanner-centered spherical coordinates to ease processing in later stages.[1]

2.2 Matching and Meshing

For motion analysis to work, vertices in one mesh must match corresponding vertices in the other. This would be trivial if all data sets were sampled from exactly the same directions, but different scans may use differing areas of interest and resolution, meaning that the advantage of a regular grid input is lost at this point.

We calculate the spherical coordinate bounds of the two meshes and use it to calculate the raster size of the final mesh. Two floating point frame buffers which will be used for further processing are created at this size. Linearization of the data and is then performed by splatting points at one of the floating point buffers, starting with single pixel points and followed by successively larger points. Depth testing is used to avoid overwrites of pixels that have already been set at smaller sizes.

At this stage the offscreen framebuffer contains only valid points located at mostly correct positions. Some pixels will contain duplicate values from when larger points were drawn. To remove duplicates and place all points in a regular grid, the frame buffer is copied to a vertex buffer and used to draw a triangle mesh. Interpolation over the triangles will give all covered points a linearly weighted depth corresponding to their position in the mesh. Duplicate points will result in degenerate triangles and simply disappear.

Some cleanup is also performed—the points created here could be used as a mesh directly, but drawing triangles blindly will result in artifacts where surfaces are disconnected as triangles connecting the edge verices will still be drawn. Such triangles are eliminated by alpha testing—the test is set to $\alpha > 0$ and vertices located near discontinuities in the mesh have their α set to 0. Interpolation will cause triangles near discontinuities to have $\alpha \in (0,1)$, which will still keep them visible. Only triangles where all vertices have $\alpha = 0$ will be invisible.

Some errors are always introduced in the scanning so the penultimate step is a smoothing of the mesh, performed by simple averaging of adjacent depth values.

Finally, all points are converted to cartesian coordinates and normals are calculated.

3 Display

In order to display differences, vertex buffers from two meshes are bound simultaneously and a vertex shader is used to interpolate or extrapolate a coordinate from the provided two. Additionally, a fragment shader samples a color ramp texture to provide color based on the distance between the two meshes.

[1] Points are stored in cartesian coordinates in all formats supported by the capture software used.

Figure 1 displays two different scans of the same door. The scans use slightly differing areas of interest and the door is ajar in scan number two. Image 3 displays color-coded and slightly exaggerated difference in the area covered by both scans. This is the intended main display mode in the final application.[2]

4 Performance

Graphics card locality is a major concern for performance. With its drastically higher throughput we wish to offload as much work as possible to the GPU, but reading this data back to the host is a bottleneck. The current solution can be executed entirely on the graphics card, given a card and drivers that support render-to-vertex-array. Unfortunately, such drivers are not yet available for the Radeon 9800 used in development so we have to copy data from card to host and back again at multiple points during execution.

Only one stage in the vertex processing is currently performed on the CPU—the cartesian to polar coordinate conversion. A bounding box needs to be calculated from the polar coordinates, which is far easier to do on the CPU, leading to a choice between performing the conversion directly on the CPU or on the GPU followed by a readback and loop for bounds calculation. The ideal solution would perhaps be to calculate bounds on the GPU so that only a very small readback is required.

The source data for the door displayed in Figure 1 contains 149 211 points. The application generates a mesh from the point set in 0.22 seconds in its current state—crippled by unnecessary readbacks.

5 Discussion and Future Work

The weakest point is currently that polygons are generated between disjoint surfaces. While these are never seen, fragments need to be generated and processed for them. A solution would be to generate new index sets that omit these triangles. Ideally, this would be performed entirely by the GPU so that data must never be read back to the host. Implementing this will likely be the next improvement made.

Overall, the system works well for the intended use—it is optimized for rapid mesh generation from two point sets and subsequent comparative display of these.

References

AMENTA, N., CHOI, S., DEY, T. K., AND LEEKHA, N. 2002. A simple algorithm for homeomorphic surface reconstruction. *International Journal of Computational Geometry and Applications 12*, 1-2, 125–141.

BAJAJ, C. L., BERNARDINI, F., AND XU, G. 1995. Automatic reconstruction of surfaces and scalar fields from 3D scans. *Computer Graphics 29*, Annual Conference Series, 109–118.

CURLESS, B., AND LEVOY, M. 1996. A volumetric method for building complex models from range images. *Computer Graphics 30*, Annual Conference Series, 303–312.

[2] The door is used for testing and is not related to the final application.

Towards rapid urban environment modelling

Ulf Söderman, Simon Ahlberg, Åsa Persson, Magnus Elmqvist
FOI (Swedish Defence Research Agency)
P.O. Box 1165, SE-581 11, Linköping, Sweden
{ulfso, simahl, asaper, magel}@foi.se

Abstract
This paper will give a brief introduction to airborne laser scanning. We will also give an overview of the current research activities at FOI Laser Systems in the fields of laser data processing and environment modelling.

Introduction
Modern airborne laser scanners (ALS) and digital cameras provide new opportunities to obtain detailed remote sensing data of the natural environment. This type of data is very suitable as basis for the construction of high fidelity 3D virtual environment models. There are many applications requiring such models, both civil and military. Applications relying on 3D-visualization, e.g. visual simulation, virtual tourism, etc. are perhaps the most common ones but there are many other important applications such as urban and environment management, crisis management, spatial decision support systems, command and control, mission planning, etc.

To support these applications, new methods for processing ALS and camera data, extracting geographic information and supporting virtual environment modelling are needed. Applications such as disaster relief management, tactical mapping, etc. may also be supported depending on the turnaround time from data to information and models (McKeown, et al 1996).

In our work, the long term goal is the development of new methods for rapid and highly automatic extraction of geographic information to support the construction of high-fidelity 3D virtual environment models from remote sensing data. As remote sensing data we have used data from recent high resolution airborne laser scanners and digital cameras. Today, airborne laser scanning is a successful and established technology for terrain surveying (Wehr & Lohr, 1999). The integration of ALS and modern digital cameras for simultaneous data collection is also rapidly increasing. ALS systems are operated both from helicopters and airplanes today and measure the position, (x, y, z) of those points where the laser pulse is reflected from the terrain surface. The resulting data from laser scanner surveys are usually data sets consisting of a large number of irregularly distributed points representing a surface model of the survey area. Figure 1 below shows the laser point distribution from a forest area (top) and from a small urban area (bottom). Laser scanning and laser data processing are active and rapidly growing areas of research (ISPRS 1999, Annapolis 2001, Dresden 2003,)

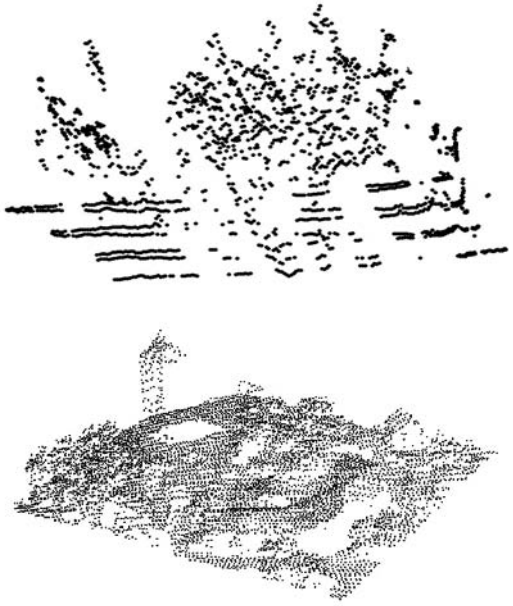


Figure 1: Laser point clouds.

Overview of developed methods
For the purpose of automatic generation of environment models, we have developed several novel methods for processing ALS data. For modelling the ground surface, we have developed a method based on active contours. The active contour is implemented as an elastic net that is pushed onto the laser data from below. Since the net has elastic forces, it will stick to the laser points that belong to the ground (Elmqvist, 2002).

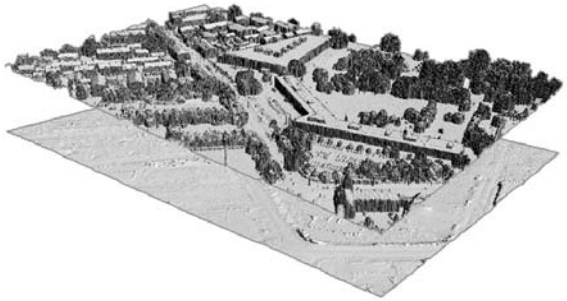


Figure 2: Ground modelling.

The remainder of the laser points (ie the points which are not classified as “ground”) can then be segmented into a number of classes. With our methods it is possible to

separate eg. ground, vegetation, buildings, roads, lamp posts and power lines. For all these classes, further data processing can be performed to extract more feature parameters. For vegetation we have developed methods for identifying the position, width, height and in many cases even the species of individual trees (Persson & Holmgren 2002), see Figure 3. The tree identification method has been verified using a ground truth dataset in collaboration with the Swedish University of Agriculture in Umeå.

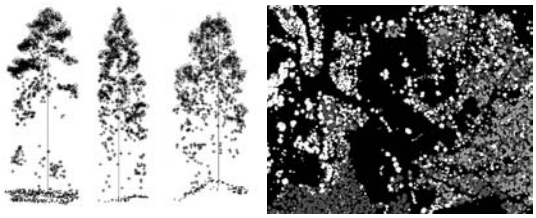


Figure 3: Laser point distribution within trees (left), tree species classification (right)

For each segment that is classified as building, i.e. the building footprint, the elevation data is used to extract planar roof faces. Next, the relationships between the roof faces are analysed. Topological points are inserted where the face's neighbours change. Sections between these points are defined as intersections, edges or both. A topological analysis is performed, where new points may be added and positions of points may be adjusted. In order to obtain building models with piecewise linear walls, the noisy edge sections are replaced by straight lines estimated using the 2D Hough transform. New points are also inserted at the intersections between the straight lines. Using these structures, 3D models of the buildings can be constructed, as illustrated in Figure 4 below.

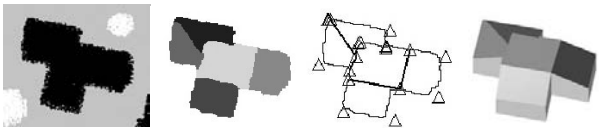


Figure 4: Building reconstruction process: Classification, data point extraction, topological analysis, 3D model.

The output form all methods can be integrated in eg. commercial terrain database generation packages or in GIS applications to create a synthetic representation of the environment (Ahlberg & Söderman, 2002). This is illustrated in Figure 5.

The data processing methods mentioned in this paper are automatic. This is a necessary requirement since the long term goal is to create an automated process from data collection to high resolution environment models.

Future work includes improved classification and further refinement of the data processing methods.

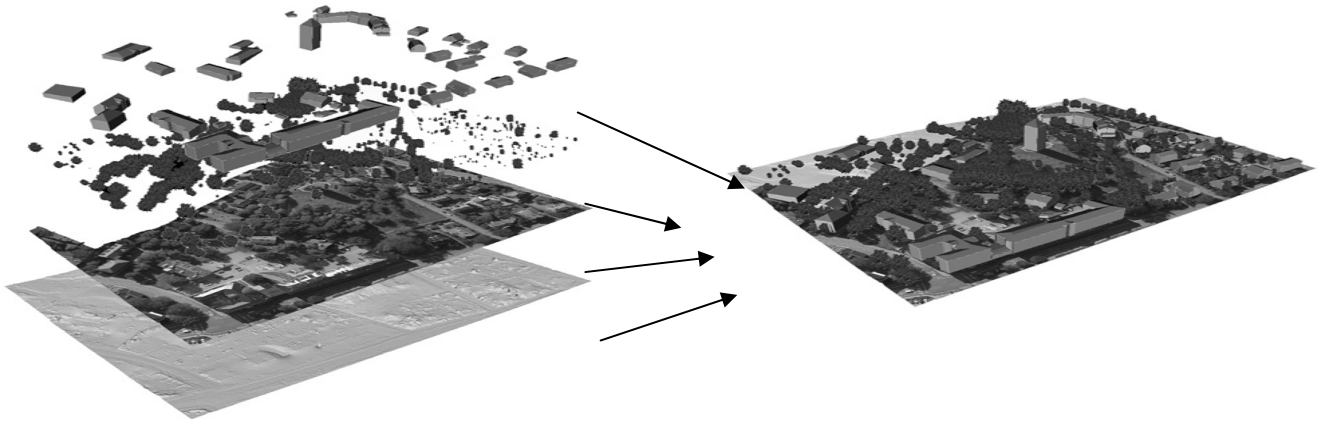


Figure 5: Extracted terrain features, an orthophoto mosaic and a DTM are integrated to create an environment model.

References

MCKEOWN, D., GIFFORD, S., POLIS, M., MCMAHILL, J. & HOFFMAN, C. 1996. Progress in Automated Virtual World Construction. *Proceedings of the 1996 ARPA Image Understanding Workshop*, Palm Springs.

WEHR A., LOHR U. 1999. Airborne Laser Scanning – an Introduction and Overview, *ISPRS Journal of Photogrammetry & Remote Sensing* vol 54 pp. 68–82

ISPRS Journal of Photogrammetry & Remote Sensing (Special issue) vol 54, issue 2/3 1999.

Proceedings of the ISPRS workshop "Land surface mapping and characterization using laser altimetry", vol XXXIV part 3/W4, ISSN 0246-1840, Annapolis, USA, October 2001.

Proceedings of the ISPRS workshop "3D reconstruction from airborne laserscanner and InSAR data", vol XXXIV part 3/W13, ISSN 1682-1750, Dresden, Germany, October 2003.

ELMQVIST M. 2002. Ground Surface Estimation from Airborne Laser Scanner Data Using Active Shape Models. *IAPRS volume XXXIV, part 3A, commission III,* pp. 114-118.

PERSSON, Å., HOLMGEN, J. 2004. Identifying Species of Individual Trees Using Airborne Laser Scanner. *Remote Sensing of Environment. Volume 90, Issue 4, 30 April 2004, Pages 415-423.*

AHLBERG, S., SÖDERMAN, U. 2002.From Environment Data Collection to High Resolution Synthetic Natural Environments, I/ITSEC 2002, Orlando, Florida, USA.

3D Reconstruction From Non-Euclidian Distance Fields

Anders Sandholm* Ken Museth†

Graphics Group @ Linköpings University‡

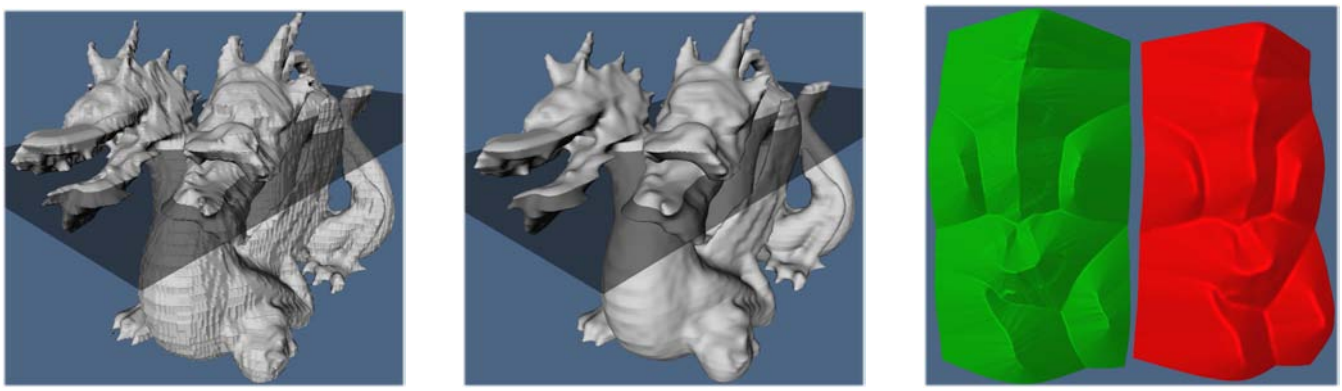


Figure 1: Left: Naive reconstruction by linear interpolation of 2D Euclidian distances to aliased (i.e. binary) input contours of a two-headed dragon. Note the semi-transparent image plane corresponding to one of the horizontal sampling planes. Middle: Our improved spline interpolation of Non-Euclidian distances. Right: Hight-field representations of the two distance fields shown as semi-transparent planes. Green is the Euclidian distance and red is the non-Euclidian distance. Note how the smoothing suppresses the medial-axis on the red surface.

Introduction

We present preliminary results on a robust method for 3D reconstruction from binary images of contours that sparsely sample 3D geometry. Our approach consists of the following steps: First Euclidian signed distance, $\Phi(x,y)$, is computed to each of the input contours. Next we smooth these distance fields by convolution with a filter kernel. The resulting non-Euclidian images are then interpolated to create a uniform volumetric implicit representation of the geometry which is finally rendered indirectly by different mesh extraction techniques.

Step 1: Deriving signed Euclidian distances

First we solving the Eikonal equation, $|\nabla\Phi| = 1$, subject to the boundary condition $\Phi(x,y) = 0$ for $(x,y) \in$ the corresponding contour. The Fast Marching Method[Sethian 1996] is a very efficient algorithm to solve such Hamilton-Jacobi equations. This produces implicit representations of the discrete contours in the whole image plane. In regions of the image plane where a pixel is equidistant from at least two other pixels on a contour $\nabla\Phi$ is undefined and one has a so-called medial-axis, see green figure. Consequently there typically only exist weak solutions to the Eikonal equation which leads to the fact that the distance transform of the contours is only Lipschitz continuous (i.e. not C1 everywhere). The presence of such medial-axis singularities in the derivative typically create noticeable kinks in 3D reconstructions directly from the Euclidian fields. To address this problem our next step is to smooth the distance fields.

Step 2: Filtering the Euclidian distance fields

As explained above we need to suppress artifacts from the presence of medial-axis in the signed Euclidian distance fields. Since the input contours are binary we also need to anti-alias the corresponding implicit representation. This all amounts to applying a smoothing filter on the Euclidian distance fields. We have experimented with the following different filter kernels:

- Uniform and anisotropic Gaussian filters.
- Adaptive Gaussian filter where the width is a functions of $|\phi(x,y)|$, i.e. the Euclidian distance to the contour.
- Bi-Laplacian filter kernel, i.e. $\Delta^2\Phi$ is minimized.

Step 3: Interpolation and mesh extraction

Once we have computed smoothed non-Euclidian distance fields we can produce an implicit 3D representation of the corresponding geometry by simple 1D interpolation between the 2D images to produce a uniform volume. The only constraint is that input contours embedded in the 2D image have to overlap to produce connected components in the 3D reconstruction. We have experimented with the following different interpolation techniques:

- Simple linear interpolation.
- Natural Cubic Spline interpolation.
- Monotonicity constrained interpolation.

As the final step in our reconstruction scheme we use mesh extraction techniques on the volume to produce a final polygonal model. We are experimenting with both Marching Cubes[Lorenson and Cline 1987] and an Extended Marching Cubes[Kobbelt et al. 2001].

Result

Figure 1 shows our reconstruction technique on horizontal slices of a two-headed dragon. Note that the contours in the slices are binary and deliberately under-sampled by a factor of five. As can be surmised from this figure our approach shows very promising results, and we plan to further develop and exploit these ideas in several exciting graphics applications.

References

KOBBELT, L. P., BOTSCH, M., SCHWANECKE, U., AND SEIDEL, H.-P. 2001. Feature sensitive surface extraction from volume data. *SIGGRAPH*, 57–66.

LORENSON, W., AND CLINE, H. 1987. Marching Cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH*, 163–169.

SETHIAN, J. 1996. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Science 93*, 4, 1591–1595.

*e-mail: andsa344@student.itn.liu.se

†e-mail: museth@acm.org

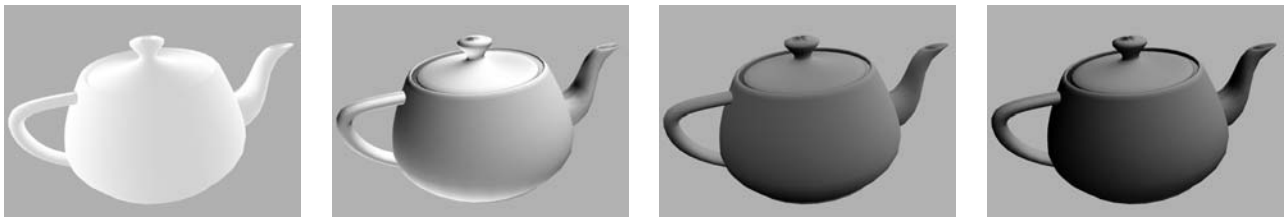
‡http://www.gg.itn.liu.se

Improved Diffuse Anisotropic Shading

Anders Hast *
Creative Media Lab
University of Gävle

Daniel Wesslén †
University of Gävle

Stefan Seipel ‡
University of Gävle



Maximum normal Proposed method

Figure 1: Comparison between diffuse light calculated with maximum normal and proposed method for two directions of anisotropy.

Abstract

Cloth, hair, brushed metal, and other surfaces with small, regular surface features exhibit anisotropic reflectance properties for which common isotropic shading methods are not suited. Shading of such materials is often implemented by computing the normal giving the maximum light contribution instead of solving the integral that is the sum of all reflected light. In this paper we show that this integral can be simplified if the direction to the viewer and fibre geometry is not taken into account. Still, this will give a more accurate result than the very rough simplification of using the maximum contribution. This computation is simple for diffuse light. However, the specular light still needs some more elaboration to work.

Keywords: anisotropic shading, diffuse light

1 Introduction

Anisotropic shading is a shading technique that could be used for materials like hair and fabrics like silk [Banks 1994]. Such materials have very small fibres with a main direction.

Poulin and Fournier [Poulin 1990] proposed a model for such materials that is rather complex and have so far been too expensive to implement for real-time applications. Even with the processing power available in graphics hardware today, a simpler model is still required.

One such simplified and frequently used model proposes that the light reflected by fibres is computed using the normal vector that results in the largest contribution of reflected light [Heidrich 1998]. This normal is obtained by projecting the light vector along the tangent onto the normal plane, which is the plane spanned by the surface normal and binormal. For the diffuse light the maximum normal is computed as

$$\mathbf{n_{max}} = \frac{\mathbf{l} - \mathbf{t}(\mathbf{l}\cdot\mathbf{t})}{\| \mathbf{l} - \mathbf{t}(\mathbf{l}\cdot\mathbf{t}) \|}. \quad (1)$$

Poulin and Fournier compute the total light reflected from the fibre. Moreover they take geometry and viewer position into account. The total light can be computed by summing the dot products using an integral, and this integral is modified by a factor depending on the viewer position. None of these considerations are taken into account in the simple model.

2 Proposed Method

We propose a trade-off which will sum the light while not taking geometry and viewer position into account. The surface normal and binormal can be used as an orthogonal base for computing any unit normal in the normal plane. This can be expressed as

$$\mathbf{n}' = \mathbf{n}\cos\theta + \mathbf{b}\sin\theta. \quad (2)$$

However, the resulting integral will be simpler by using $\mathbf{n_{max}}$ and $\mathbf{n_{min}}$ as a base. Figure 2 illustrates these vectors.

The normal contributing least to the light is the one where $\mathbf{n_{min}}\cdot\mathbf{l} = \mathbf{0}$, it is obtained by

$$\mathbf{n_{min}} = \frac{\mathbf{l}\times\mathbf{n_{max}}}{\| \mathbf{l}\times\mathbf{n_{max}} \|}. \quad (3)$$

The minimum of a cosine function is offset by $\pi/2$ from the maximum. This is excatly the case for $\mathbf{n_{min}}$ and $\mathbf{n_{max}}$, hence these can be used as a base. The light is computed by

$$\Phi = \frac{1}{\pi}\int_0^{\rho} I_{\min}\cos\theta + I_{\max}\sin\theta\,\mathrm{d}\theta, \quad (4)$$

$$\rho = \cos^{-1}(\mathbf{n_{min}}\cdot\mathbf{b}). \quad (5)$$

Light is integrated over the portion of the fibre that would be visible if occlusion occured from the surface plane and not from other

*e-mail: aht@hig.se
†e-mail: dwn@hig.se
‡e-mail: ssl@hig.se

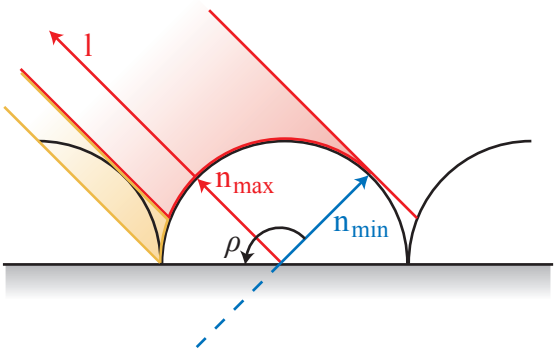


Figure 2: Lit area and integration base vectors.

fibres, and then divided by π as if no occlusion occured. Figure 2 illustrates the integration, the red area is actually lit but the red and yellow areas are integrated.

The integral in (4) is very similar to what Poulin and Fournier propose, however

$$I_{\min} = \mathbf{n_{min}} \cdot \mathbf{l} = 0 \tag{6}$$
$$I_{\max} = \mathbf{n_{max}} \cdot \mathbf{l}. \tag{7}$$

The equation can therefore be reduced to

$$\Phi = \frac{1}{\pi} \int_0^{\rho} I_{\max} \sin\theta \, d\theta, \tag{8}$$

evaluation of which leads us to

$$\Phi = \frac{1}{\pi} \left(-I_{\max} \cos\rho + I_{\max} \cos 0 \right). \tag{9}$$

However, from (5) we have $\rho = \cos^{-1}(\mathbf{n_{min}} \cdot \mathbf{b})$, so the final formulation is

$$\Phi = \frac{1}{\pi} (\mathbf{l} \cdot \mathbf{n_{max}})(1 - \mathbf{n_{min}} \cdot \mathbf{b}). \tag{10}$$

In this form all trigonometric functions are eliminated, leaving only normalization, dot- and cross product, all of which are efficient operations in fragment shaders on current hardware.

3 Discussion

The proposed method is a compromise between accuracy and speed —far more accurate than and the simplistic maximum normal method but faster and less accurate than the Poulin method.

Which method is most suitable naturally depends on the situation, but we have presented a reasonable compromise in cases where performance is required and the maximum normal method is not enough.

Figure 1 compares our method with maximum normal lighting. Keep in mind that only diffuse light is used.

Research continues in search of a comparable method for the specular component and a more accurate complete solution.

References

D. BANKS 1994. *Illumination in Diverse Codimensions.* In Proceedings SIGGRAPH, pp. 327–334.

W. HEIDRICH, H-P. SEIDEL. 1998. *Efficient rendering of anisotropic surfaces using computer graphics hardware.* In Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP), 1998.

P. POULIN, A. FOURNIER 1990. *A model for anisotropic reflection.* ACM SIGGRAPH Computer Graphics, Proceedings of the 17th annual conference on Computer graphics and interactive techniques, Volume 24 Issue 4, pp. 273–282.

An Optimized, Grid Independent, Narrow Band Data Structure for High Resolution Level Sets

Michael Bang Nielsen*
University of Aarhus

Ken Museth†
GG‡@Linköpings University

1 Introduction

Level sets have recently proven successful in many areas of computer graphics including water simulations[Enright et al. 2002] and geometric modeling[Museth et al. 2002]. However, current implementations of these level set methods are limited by factors such as computational efficiency, storage requirements and the restriction to a domain enforced by the convex boundaries of an underlying cartesian computational grid. Here we present a novel very memory efficient narrow band data structure, dubbed the Sparse Grid, that enables the representation of grid independent high resolution level sets. The key features our new data structure are

- Both memory usage and computational efficiency scales linearly with the size of the interface.
- The values in the narrow band can be compressed using quantization without compromising visual quality.
- The level set propagation is independent of the boundaries of an underlying grid. Unlike previous methods that use fixed computational grids with convex boundaries our Sparse Grid can expand and/or contract dynamically in any direction with non-convex boundaries.
- Our data structure generalizes to any number of dimensions.
- Our flexible data structure can transparently be integrated with the existing finite difference schemes typically used to numerically solve the level set equation on fixed uniform grids.

2 Data Structure

Previously proposed methods for localizing level set computations[Peng et al. 1999]require the entire 3D grid and additional data structures representing the narrow band to be present in memory. However, our Sparse Grid data structure implements localized level set computations on top of a dynamic narrow band storage scheme to obtain both a time and space-efficient data structure.
Figure 1 shows a human head and a slice of its corresponding 3D Sparse Grid representation. The values of all grid points in the narrow band, shown in red and green, are explicitly stored and may additionally be quantized to further reduce the memory footprint. The (x,y,z) index-vector of each grid point in the narrow band must also be stored, however explicit storage does not scale well. Instead we exploit the connectivity of the narrow band to develop an efficient index-vector storage scheme with a compression. This compact storage scheme is defined recursively in the dimensions of the grid and allows the Sparse Grid to easily generalize to any dimension. For a 3D Sparse Grid, the scheme stores: 1) The start and end z-index of each connected component in the z-direction. 2) The start and end y-index of each connected component in the y-direction contained in the projection of the 3D narrow band onto the x-y plane. 3) The start and end x-index of each connected component in the x-direction contained in the recursive projection of the 3D narrow band onto the x-axis. The actual grid point index-vector of the values in the narrow band are decoded on the fly. Sequential access to all grid points in the Sparse Grid can be done in linear time, thus providing constant time access to each grid point on average. Level set propagations typically require information about the gradient, curvature etc. in each grid point, which requires knowledge of neighboring points. Constant time access to neighboring grid points within a stencil is provided by iterating a stencil of iterators through the narrow band. Random access is logarithmic.
The N-dimensional Sparse Grid enables the narrow band to be rebuilt in linear time using a fast algorithm that takes advantage of the information about connected components readily available in the storage format.

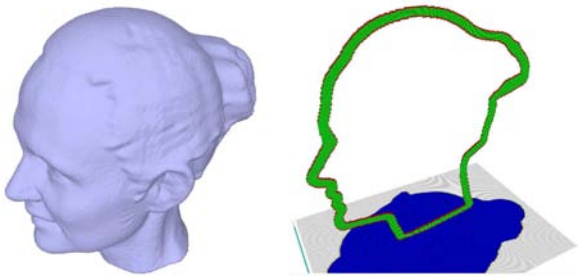


Figure 1: Human head and slice of Sparse Grid representation.

3 Evaluation

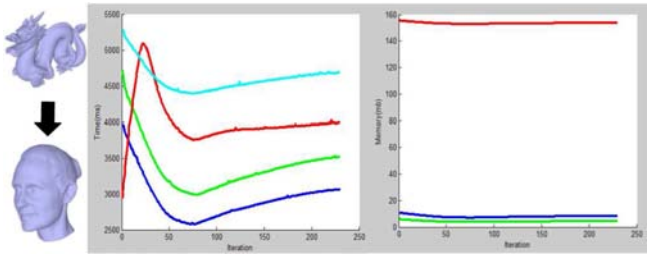


Figure 2: Morph time and memory usage on a $193 \times 356 \times 251$ grid.

Figure 2 shows the time (left) and memory (right) usage for a level set morph computed with the method of Peng(cyan), the method of Peng improved with a memory efficient localized narrow band update (red), the Sparse Grid (blue) and the Sparse Grid with a 16 bit quantization (green). The Sparse Grid methods are mostly faster and use far less memory. The methods of Peng use the same amount of memory. The improved method of Peng (red) does not ensure cache coherency which explains the rapid increase in running time at the beginning. We plan to apply the Sparse Grid to several research areas in computer graphics in the future.

References

ENRIGHT, D., MARSCHNER, S., AND FEDKIW, R. 2002. Animation and rendering of complex water surfaces. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 736–744.

MUSETH, K., BREEN, D. E., WHITAKER, R. T., AND BARR, A. H. 2002. Level set surface editing operators. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 330–338.

PENG, D., MERRIMAN, B., OSHER, S., ZHAO, H., AND KANG, M. 1999. A pde-based fast local level set method. *J. Comput. Phys. 155*, 2, 410–438.

*e-mail: bang@daimi.au.dk
†e-mail: museth@acm.org
‡http://www.gg.itn.liu.se

The Virtual Forest

Daniel Wesslén*
University of Gävle

Stefan Seipel†
University of Gävle

Figure 1: Screenshots from two different forest scenes.

1 Introduction

Offline rendering of computer generated trees has received much research in the last decade, resulting in some impressive imagery [Deussen et al. 1998]. Realtime rendering for simple cases has also been explored, both for highly detailed trees viewable from a distance [Weber and Penn 1995] and the textured low-polygon versions seen in current games.

We present a system that renders an animated synthetic forest in realtime for interactive walkthroughs. Up close, trees are detailed enough that individual leaves are animated, while continuous level-of-detail adjustments keep processing load low when viewed from afar.

2 Creation

Tree creation is based on the algorithm described by Weber and Penn [1995] but adapted to better suit nordic trees such as spruce and pine, cleaned from many implementation artifacts, and generally made easier to handle for tree designers and developers.

The most important enhancement to tree creation is the addition of a health parameter, which is required for modeling of realistic pine trees. Moderate reductions in health cause leaves to be randomly omitted from the creation process. Larger reductions remove branches as well.

3 Display

A continuous level-of-detail algorithm keeps rendering running smoothly independently of the distance from which trees are viewed. Up close, individual leaves and branches are rendered as textured primitives. When the camera moves further away these are gradually replaced by simple points and lines. Moving even further, they are removed when the average projected size of the primitives fall below one pixel.

Display differs from [Weber and Penn 1995] in two important aspects—the original algorithm had to traverse the entire tree to select which parts of it were to be rendered, and leaves were singly-colored complex shapes. We use prearranged vertex buffers and select parts of the tree by computing index ranges in these. Leaves are rendered as textured quads, improving storage efficiency and detail. Shapes may be applied to the leaves by alpha testing.

Entire trees are animated in a simple swaying motion around the base by the vertex shader and leaves are individually animated in a rotation around their parent stem. While this is simplistic, the results are pleasing.

Shaders also increase the diversity of the forest by allowing slight changes to the size and color of tress instanced from the same base model and by reducing memory requirements due to per-tree rather than per-vertex storage of some parameters.

4 Results

Figure 1 contains screenshots from our simulation, these were captured at 1600×1200, running in excess of 50 frames per second on a Radeon 9800 XT. Since essentially all work is performed by the graphics card the CPU is irrelevant during rendering.

The system allows for realtime frame rates on affordable consumer cards for relatively complex scenes.

References

DEUSSEN, O., HANRAHAN, P., LINTERMANN, B., MĚCH, R., PHARR, M., AND PRUSINKIEWICZ, P. 1998. Realistic modeling and rendering of plant ecosystems. *Computer Graphics 32*, Annual Conference Series, 275–286.

WEBER, J., AND PENN, J. 1995. Creation and rendering of realistic trees. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM Press, 119–128.

*e-mail: dwn@hig.se
†e-mail: ssl@hig.se