Proceedings of the

18th Central European Seminar on Computer Graphics

May 25 - 27, 2014
Smolenice, Slovakia
Co-organized with SCCG

Institute of Computer Graphics and Algorithms
Vienna University of Technology

Faculty of Mathematics, Physics and Informatics
Comenius University Bratislava

Sponsors

Edited by Michael Wimmer, Jiří Hladůvka, and Martin Ilčík © 2014
ISBN:978-3-9502533-6-8

Impressum

Vienna University of Technology
Institute of Computer Graphics and Algorithms
Favoritenstraße 9-11/186
1040 Vienna

ISBN 978-3-9502533-6-8

Welcome to CESCG 2014!

This book contains the proceedings of the 18th Central European Seminar on Computer Graphics, short CESCG, which continues a history of very successful seminars. Again this year, CESCG proceedings have an ISBN (978-3-9502533-6-8) and will therefore remain retrievable as long as there are libraries!

The long history of CESCG has started in 1997 in a medium-sized lecture room in Bratislava, bringing together students from Bratislava, Brno, Budapest, Graz, Prague, and Vienna. The idea found wide appraisal and the seminar moved to the beautiful castle of Budmerice, where it was held for 8 consecutive years, constantly growing in size and attraction. It was just in the 10th anniversary year 2006 that CESCG had to take a detour to move to Častá-Papiernička Centre, while it was back in Budmerice castle since 2007. Unfortunately, since 2011 the Budmerice castle is not available for scientific activities. After spending the one year in Viničné, in 2012 we moved to the beautiful castle in Smolenice.

Who are the CESCG heroes who made this year's seminar happen? In no particular order – because many people were involved equally – we would like to thank the organizers from Vienna: **Michael Wimmer**, Anita Mayerhofer, Werner Purgathofer, Katharina Krösl and Bernhard Steiner. Special thanks goes to **Martin Ilčík** for taking care of the complete reviewing process and scientific program preparation. We are very thankful to the CESCG organizers from Bratislava, mainly **Andrej Ferko**, always an inspiration to CESCG; and Ela Šikudová, Janka Běhal Dadová, David Běhal and Ivka Varhaníková for the excellent preparations and on-site organization.

The main idea of CESCG is to bring students of computer graphics together across boundaries of universities and countries. Therefore we are proud to state that we have achieved again a very high number of 11 participating institutions and a very tight time schedule of 20 valuable student works and two invited talks. We welcome groups from Bratislava (UK and STU), Slovakia; Brno (VUT and MU) and Prague (CTU), Czech Republic; Budapest, Hungary; Graz and Vienna (TU), Austria; Szczecin, Poland; and Maribor, Slovenia.

We assembled a large International Program Committee of 16 members, allowing us to have each paper reviewed by three IPC members during the informal reviewing process. We would like to thank the members of the IPC for their contribution to the reviewing process. The IPC of CESCG 2014 consists of:

Jiří Bittner
Silvester Czanner
Andrej Ferko
Jasminka Hasić
Ivana Kolingerová
Radosław Mantiuk
Selma Rizvić
Michael Schwärzler
Jiří Sochor
Markus Steinberger
Marc Streit
László Szirmay-Kalos
Ania Tomaszewska
Michael Wimmer
Borut Žalik
Pavel Zemčík

The first invited talk "The Role of Perception in Graphics" will be held by Rafał Mantiuk from Research Institute of Visual Computing of the School of Computer Science of Bangor University, United Kingdom. The second invited talk by Roberto Scopigno from Visual Computing Lab of the Institute of Information Science and Technology of National Research Council of Italy, will be about "Visual Media for Cultural Heritage: An Opportunity for Assessing, Finding Limitations and Enhancing Technologies".

The seminar is is co-organized with the Spring Conference on Computer Graphics (SCCG), which takes place right after the seminar.

The organization of a seminar where there are only low expenses for the students requires funding. We are very thankful to the sponsors of CESCG 2014:

- NVidia, The Way It's Meant to Be Played,
- VRVis Research Center,
- OCG, The Austrian Computer Association,
- SISp, Slovak Society for Computer Science,
- Eurographics, The European Association for Computer Graphics.

The best paper will be awarded by an NVidia Shield console for development of next-generation Android games and wirelessly streamed PC games.

Please note that the electronic version of these proceedings is also available at http://www.cescg.org/CESCG-2014/.

April 2014,

Michael Wimmer
Jiří Hladůvka
Martin Ilčík

Table of Contents

Invited Talks

Augmented Reality

Parallel Graphics

Computer Vision

Lighting and Natural Phenomena

Geometry Processing

Attention and Metrics

Sponsors of CESCG 2014

Invited Talks

The Role of Perception in Graphics

Rafał Mantiuk

Bangor University
United Kingdom

Abstract

Today's computer graphics techniques make it possible to create imagery that is hardly distinguishable from photographs. However, a photograph is clearly no match to an actual real-world scene. I argue that the next big challenge in graphics is to achieve perceptual realism by creating artificial imagery that would be hard to distinguish from reality. This requires profound changes in the entire imaging pipeline, from acquisition and rendering to display, with the strong focus on visual perception.

In this talk I will give an overview of two projects that demonstrate the role of the visual perception in graphics. In the first project we integrated eye-tracking with real-time rendering to improve the accuracy of an eye-tracker and to enhance the rendering using the gaze data. The much improved eye-tracking accuracy let us use gaze-data in applications that have not been possible before, such as gaze-contingent simulation of the depth-of-field effect or a gaze-contingent heads-up display. In the second project we created a new model of the colour and luminance perception across the wide range of luminance, accounting for both night and day-light vision. The model let us simulate the appearance of night scenes on regular displays, or generate compensated images that reverse the changes in vision due to low luminance levels. Such a simulator of visual perception can be used in games, driving simulators, or as a compensation for displays used under varying ambient light levels.

Bibliographical Details

Rafał Mantiuk is a senior lecturer (associate professor) at Bangor University (UK) and a member of a Reasearch Institute of Visual Computing. Before comming to Bangor he received his PhD from the Max-Planck-Institute for Computer Science (2006, Germany) and was a postdoctoral researcher at the University of British Columbia (Canada). He has published numerous journal and conference papers presented at ACM SIGGRAPH, Eurographics, CVPR and SPIE HVEI conferences, applied for several patents and was recognized by the Heinz Billing Award (2006). Rafal Mantiuk investigates how the knowledge of the human visual system and perception can be incorporated within computer graphics and imaging algorithms. His recent interests focus on designing imaging algorithms that adapt to human visual performance and viewing conditions in order to deliver the best images given limited resources, such as computation time or display contrast.

Visual Media for Cultural Heritage: An Opportunity for Assessing, Finding Limitations and Enhancing Technologies

Roberto Scopigno

National Research Council of Italy

Abstract

Digital technologies are now mature for producing high quality digital replicas of Cultural Heritage (CH) artifacts. The research results produced in the last two decades have shown an impressive evolution and consolidation of the technologies for acquiring high-quality digital 3D models, encompassing both geometry and color (or, better, surface reflectance properties); technologies for the interactive visualisation of complex models and the integration of different media have been also an important subject of research.

In this talk, I will present the more recent progresses, focusing on practical solutions which aim at a major impact in real applications. The talk will also try to give a glance into the near future, demonstrating how geometry processing and visualization could become a major instrument in the study and dissemination of our cultural heritage.

Bibliographical Details

Roberto Scopigno is a Research Director at ISTI, an Institute of the Italian National Research Council (CNR) located in Pisa, and leads the Visual Computer Lab. He graduated in Computer Science at the University of Pisa in 1984, and has been involved in Computer Graphics since then.

He is currently engaged in several EU and national research projects concerned with multiresolution data modeling and rendering, 3D digitization/scanning, scientific visualization, geometry processing, virtual reality and applications to Cultural Heritage.

He published more than two hundreds papers in international refereed journals/conferences with Google Scholar h-index 39 and more than 7100 citations. He presented invited lectures or courses at several international conferences. He was Co-Chair of several international conferences and served in the program committees of international events.

Since 2012 he is Editor In Chief of the ACM Journal of Computing and Cultural Heritage; he served as Editor in Chief of the journal "Computer Graphics Forum" (2001-2010). He is member of Eurographics, served as elected member of the Eurographics Executive Committee since 2001 and was the Eurographics Chairman on 2009-2010. He is recipient of several awards, including the EG Distinguished Career Award (2014), the EG Outstanding Technical Contribution Award (2008) and the Tartessos Virtual Archeology Award (2011).

Augmented Reality

Integrating Motion Tracking Sensors to Human-Computer Interaction with Respect to Specific User Needs

Michal Vinkler*
Supervised by: Jiří Sochor†

Faculty of Informatics
Masaryk University
Brno / Czech Republic

Abstract

This paper presents a novel framework for combining Microsoft Kinect device (Kinect) and Leap Motion sensor with Java Monkey Engine and utilizing it for natural human computer interaction. The framework supports the standard input devices (keyboard, mouse) as well as selected motion tracking sensors Microsoft Kinect, Leap Motion). To demonstrate its aplicability, the framework was subsequently used for creating the demonstration application called Labyrinth. This application was designed according to the current requirements of psychologists from the Department of Psychology at the Faculty of Arts, Masaryk University, with respect to children with specific disorders, such as ADHD (Attention Deficit Hyperactivity Disorder). Similar kinds of applications are widely utilized by psychologists for observing and evaluating their impact on various groups of patients.

To make the application more realistic, our framework integrates the stereo projection and the jBullet physics library for simulation of the gravitational force applied to the scene. The resulting application forms a comprehensible basis for future applications based on this framework.

Keywords: virtual environment, human-computer interaction, jBullet, physical simulation, Microsoft Kinect, Leap Motion, motion tracking

1 Introduction

The expansion of modern technologies in the last decades introduced the virtual environment and motion tracking to the broader community of users. Moreover, these users currently come from diverse fields, such as flight industry, robotics, art or medicine. In the latter case, many applications utilizing interactive environments were created to help both doctors during the medical intervention [2] [16] and patients during the recovery process. Another very interesting and important area is the usage of motion tracking in the process of the treatment and therapy for children with various disorders, such as Attention Deficit Hyperactivity Disorder or Oppositional Defiant Disorder.

Attention Deficit Hyperactivity Disorder (ADHD) is a psychiatric disorder of the neurodevelopmental type in which there are significant problems of attention, hyperactivity, or acting impulsively that are not appropriate for the person's age [3]. Oppositional Defiant Disorder (ODD) is described as an ongoing pattern of anger-guided disobedience, hostility, and defiant behavior toward authority figures that goes beyond the bounds of normal childhood behavior. Children suffering from this disorder may appear stubborn and often angry [10].

Studies concentrating on the influence of playing video games to different groups of users consistently show that this activity improves the hand-eye coordination and increases humans visuospatial skills. This increase of brain activity is tied not only to playing games but also to several other real-world scenarios. When the brain encounters new visual and auditory stimulation, or new and different way of processing information, it can influence the brain in the most remarkable ways [1]. This improvement may naturally mark in the person's behavior and then be incorporated into the daily life [6].

The goal of this paper was to create the environment which is straightforward to use and allows designing interactive games for children with specific needs in easy way. The resulting framework uses modern features and devices in virtual environments, enables full body interaction and can display the output using the stereoscopic projection. Moreover, the new system integrates the simulation of physical forces which supports the plausibility and perception of real environment of the implemented games. Although the Microsoft Kinect used for motion tracking is primarily designed for Microsoft Windows operating system, our framework is platform independent as it uses Java programming language and open source libraries for communication with the Kinect device. For user convenience, the detection of gestures for both hands was added.

The idea behind this project is to provide children with smaller tasks in a form of a mini-games. Children are then asked to keep playing and repeat the game to be able to achieve mastery. The system supports basic logging thus

*xvinkle1@fi.muni.cz
†sochor@fi.muni.cz

the psychologists can subsequently evaluate if there is any observable impact or improvement, for example that the child can stay focused on one activity much longer than before, etc.

For demonstration purposes we implemented a small game called Labyrinth (see Figure 5). It basically aims to strengthen the hand-eye coordination. In terms of practical application, this can help to improve real world skills, such as handwriting or any other activity requiring the fine motor activity. In the past, it was proved by various studies that children with ADHD have had difficulties with various hand-eye coordination skills [9]. When practicing, the brain can learn how to focus on what the hand is doing. With the repeated play, the brain can become more accurate and more aware of the focus required for complete control of the body.

The Labyrinth playing field is formed by the rectangular-shaped maze (see Figure 1). Player can tilt the board and therefore indirectly control the movement of the metal-like ball located inside. The maze board contains holes (so the ball can fall through) and obstacles defining the inner layout of the maze. The goal is to guide the ball through the maze to the finish hole while avoiding any other holes in the fastest possible time. There are multiple levels predesigned with difficulty ranging from very easy to hard. For evaluation purposes, selected values are logged. For example, the runtime of the application, the duration of each game, the idle time at the beginning (when the user sees the maze for the first time and has to make the decision about the route) and idle time at crossroads.

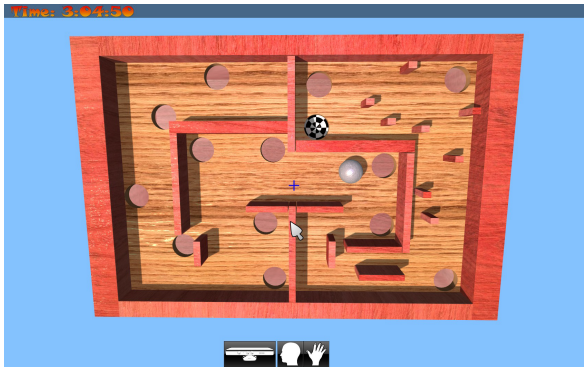


Figure 1: Example of the playing field in Labyrinth application.

2 Related Work

Many studies have been done in last years to support the hypothesis that the daily use of brain games can help to strengthen the focusing ability and the executive functioning in adolescents with ADHD. Wegrzyn et al. [18] confirmed the positive impact on participants playing Nintendo DS game Brain Age twenty minutes per day for five weeks. Hashemian and Gotsis [7] created series of minigames, each one focused on a specific strength or weakness prevalent in children with ADHD. The project uses Microsoft Kinect for a full body motion tracking.

Virtual reality (VR) is an emerging technology with a variety of potential benefits for many aspects of rehabilitation assessment, treatment, and research. Schultheis and Rizzo [15] focused on examining the specific benefits VR offers to consumers and providers of rehabilitation services. They also discuss the potential areas of application and important considerations in applying the VR technology. Strickland [17] concentrated in her study on using VR as a learning aid with an immersive headset system. Finally, Parsons et al. [14] made a controlled clinical comparsion of attention performance of children with ADHD in a VR classroom. These children exhibited more omission errors, commission errors, and overall body movement than normal control children in the VR classroom.

Virtual environment enables to utilize new interaction techniques mainly via spatial input tracking and stereoscopic rendering. From the technical point of view, various virtual reality systems were created which can be exploited by serious games. The VRECKO framework for virtual reality [5] supports several kinds of stereoscopic projection and is able to process data from various input devices. Except for the traditional devices, such as mouse and keyboard, the OptiTrack system can be used, enabling to capture the position of objects in space, 3D mouse, Nintendo Wii Remote and Microsoft Kinect.

3 System Overview

In this section we briefly describe the utilization of Java Monkey Engine for our purpose. Then the motion tracking devices used in our solution are introduced. Finally we will concentrate on the technical solution of devices integration.

3.1 Java Monkey Engine

In order to utilize the most of already implemented and tested features, our new framework was integrated into the Java Monkey Engine 3.0 (JME) [8]. JME is free, open source game engine, programmed entirely in Java, intended for wide accessibility and quick deployment. It is the high level programming engine using the underlying toolkit and graphics library for low level tasks, such as rendering 3D primitives, texturing, scene culling, etc. JME itself does not integrate any motion tracking device. We have chosen to implement the support for Microsoft Kinect and Leap Motion sensors.

3.2 Devices

Microsoft Kinect is a full body motion tracking device. By using a RGB camera and a four-element, linear micro-

phone array it can record both video and audio streams. The depth sensor transmits invisible near-infrared light and measures its "time of flight" after it reflects off the objects. Knowing how long the light takes to return, Kinect can calculate how far away an object is. As a result, an accurate depth map of the scanned area is created.

Leap Motion allows accurate movement tracking of hands, fingers or other rigid objects (for example a pen) working on a very similar principle as Microsoft Kinect. Using two monochromatic IR cameras and three infrared LEDs, the device observes a roughly hemispherical area, to the distance of about 1 meter. The LEDs generate a 3D pattern of dots of IR light which is reflected from the objects placed in the scanned area and then captured by IR cameras.

3.3 Kinect – OpenNI Framework

The OpenNI organization [13] is an industry-led, non-profit organization, formed to certify and promote the compatibility and interoperability of Natural Interaction devices, applications and middleware. The organization has made an open source multi-language, cross-platform framework available – the OpenNI framework – which provides an application programming interface (API) for writing applications utilizing natural interaction.

The main purpose of OpenNI is to form a standard API that enables communication with both physical devices (audio, video and depth sensors) and middleware components, which further process the input data. In this project, NiTE 1.3.1 Middleware was used (see Figure 2).

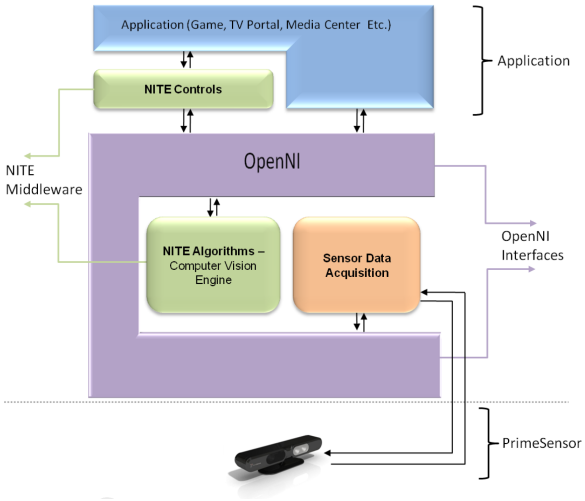


Figure 2: A three-layered view of the OpenNI concept (taken from [13]).

3.4 Kinect – NiTE Middleware

NiTE Middleware [11] was based on the concept including two base paradigms: *Point Control* and *Full Body Control*.

When *Point Control* is used, only one point (typically a hand) is being tracked and NiTE gestures for this point are recognized. There are three possible states for *Point Control*:

- Not in Session: In this state, there is no active session, hence the system is in a mode of scanning the scene to detect a "focus gesture" (for example, waving). Once this gesture is recognized, the state changes to *In Session*.
- In Session: In this state, there is a hand that is currently in control and being tracked by the system.
- Quick Refocus: This is an intermediate state, in which, while in session, the hand point is lost. We don't want to stop the session yet, but rather give a grace time period in which the session can be resumed with a different (perhaps shorter or easier) hand gesture (referred as *Quick Refocus Gesture*). While in *Quick Refocus* state, the session can also be resumed using the regular focus gesture. Once the grace period has timed out, the state changes to *Not in Session*. The *Quick Refocus* state is optional.

When *Full Body Control* is active, the whole body is being tracked. The player's skeleton data is extracted from the depth map and handed over for further processing. Therefore, the application receives information about position and rotation of up to 16 joints, together forming user's full-body skeleton.

In our framework, both controls were utilized. Based on the user experience, the focus gesture was set to waving user's hand, whereas the quick refocus gesture is defined as raising the hand. Unlike Point Control, Full Body Control does not determine any states and the flow between them. Therefore, it was necessary to define, how will the application react on the newly detected user in the scene. In order to start skeleton tracking, it is often required to strike a *Psi* pose (standing in front of the camera and holding arms up). NiTE Middleware can also be requested to perform a skeleton calibration right after new user is detected in the scene. If the calibration is successful, the tracking is started (see Figure 3).

Also, initializing Kinect and accessing the events from OpenNI framework is rather complicated. To make this process easier and to achieve full integration with the JME, the Kinect input was programmed to be handled by JME as a standard input (the similar way as for example a joystick). The initialization process is done automatically when the application is started. If not successful (the device is missing or the drivers are not installed), the application can still be controlled by standard input devices or by Leap Motion.

In order to make the tracking-related information easily accessible, new structures representing tracked joints and the whole skeleton were created. The application can ask any time for the tracked skeletons and position of their

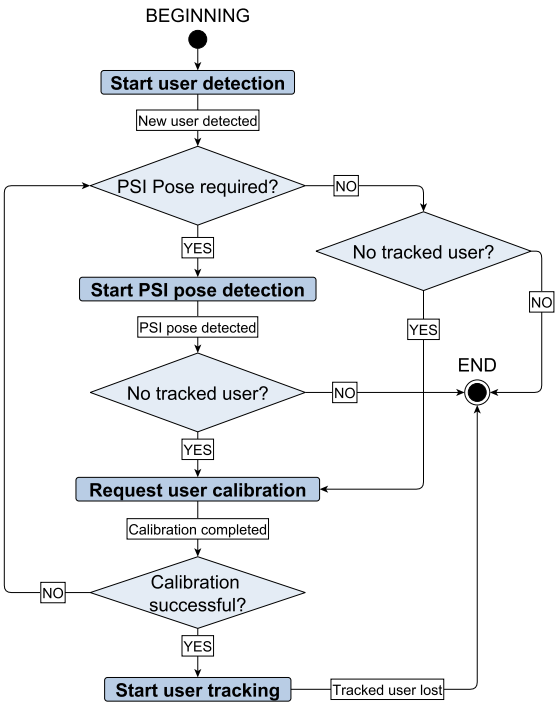


Figure 3: The procedure for new user detection in the scene.

joints. The framework also provides an interface for access to all the important OpenNI and NiTE components (for example VGA and depth image generators), and also to all NiTE middleware events.

3.5 Kinect – Gestures Detection

NiTE defines a set of basic gestures recognized for active hand in Point Control tracking. If we want to use the point tracking for controlling the selector (i.e., mouse cursor), the same hand would have to manage all the gestures as well. In our framework, we wanted to have the possibility to use both hands for controlling the application. Daria Nitescu [12] mentions several strategies to accomplish the select and click action. Based on our experience, selecting the item with one hand and clicking with the other hand is convenient and user friendly. This action however cannot be accomplished using the NiTE gesture recognition ability.

In order to provide as much freedom as possible when controlling the application, we defined a new set of custom gestures. The input data is taken from Full Body Control, which allows us to define our gestures on any limb we choose, overcoming the limitation of NiTE. Gesture manager handles all registered detectors. Whenever the input data is updated, the arm gestures detector checks if any of the defined gestures was performed and fires respective events accordingly. Let us explain how the gesture *Arm Forwards* (analogue of NiTE's *Push* gesture) works. The arm gesture detector retrieves the current position of the joints representing the user's shoulder and hand and checks if the position is valid (i.e., NiTE can recognize the joints location in the depth map). The distance between the two joints must exceed a predefined value in depth in order to activate the gesture (see Figure 4). Once the user pushed in depth sufficiently enough, he or she has to release the hand. The depth value can be set manually or estimated based on the user's height.

If necessary, we could register for example leg gestures detector (with a set of relevant gestures defined) in a similar way.

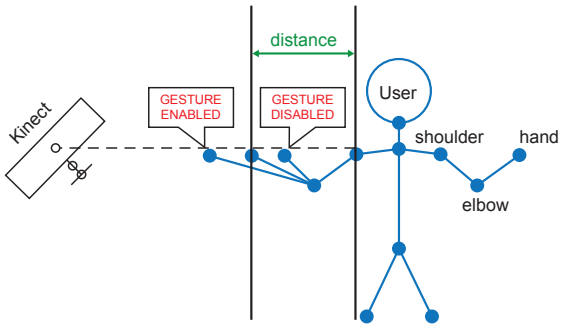


Figure 4: Schematic representation of *Arm Forwards* gesture detection.

3.6 Leap Motion

For Leap Motion integration, the official SDK was used. The whole process is rather straightforward compared to the Kinect integration. When the Controller object is created, it connects to the Leap Motion software running on the computer and makes hand tracking data available through the Frame objects. Each computed frame contains information about recognized hands and fingers of a particular hand, finger tips position, recognized gestures, hand's sphere radius, normal vector and direction and palm absolute position. The pitch, roll, and yaw can be computed from the hand's direction vector. All the information is available to the application through the Controller interface.

3.7 Physical Simulation

Our framework utilizes the jBullet, Java port of Bullet Physics Engine [4]. The jBullet is implemented into the JME as one of the application states. Therefore, it can be paused (resulting in freezing physical simulations) or resumed as needed. The main features of jBullet include: discrete and continuous collision detection, swept collision queries, ray casting with custom collision filtering, support for generic convex and non-convex triangle meshes.

3.8 Stereo Projection

JME was primarily designed for displaying monocular image. By modifying some core classes we were able to create side-by-side stereoscopic projection. It was necessary to position two views with separate cameras into the main window. Hardware cursor provided by the operating system cannot be duplicated, therefore it was replaced with two software cursors displayed in both views. The position of software cursors is calculated from the position of the hardware cursor to match the stereo projection accordingly. Also the GUI rendering was modified to appear in both views.

The two images representing the GUI projection are not mutually shifted in x-axis so they do not create the spatiality effect. However, the shift of the scene images is customizable, which directly determines the parallax type and the depth of the stereoscopic 3D effect.

4 Labyrinth Application

The objective of the sample mini-game in this project is to demand focused attention while players use various hand moves and body gestures to control the application. This process aims to teach children to pay attention for longer periods of time. The game can be fully controlled by implemented motion tracking devices, either Kinect or Leap Motion. When Kinect is used, the cursor movement and tilting of the maze is controlled by user's right hand while the custom-defined gestures (Subsection 3.5) are performed by left hand (see Figure 5). In the case of Leap Motion being used, only one hand is necessary for performing the actions mentioned above. To achieve the realistic behavior, jBullet was used for controlling the ball movement. The side-by-side stereo projection is also supported.

4.1 Events Logging

One of the main requirements from psychologists was logging of the important events in the application. The logged values are:

- Application runtime: Indicates, how much time the user spent with practicing on the daily basis.
- Duration of each game: Used for comparing of the progress.
- Time spent in the decision areas: Each maze contains the defined "decision areas" – the starting area in the maze and forks where the path splits. For each decision area the idle time spent in it is logged (see Figure 6).
- Ball falling through a hole: In both cases of ball falling either into the finish or any other hole, the time of the event and the ball position is logged.

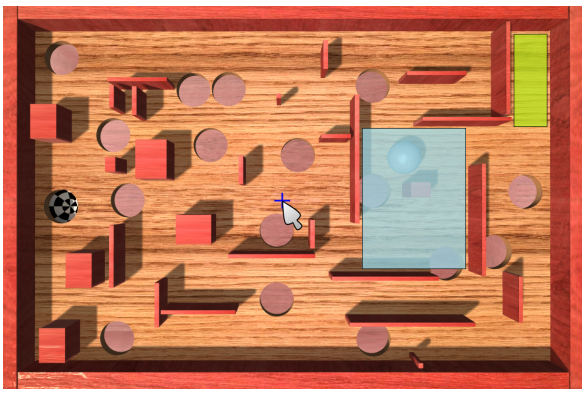


Figure 6: Example of a maze with decision areas highlighted (green - the starting area, blue - the fork).

5 Discussion

We believe that our proposed system along with the logged data will help to gain the novel insights for the psychologists. However, the evaluation of the system as well as the Labyrinth mini-game is currently in an early stage, as this preliminary version is standing at the very beginning of a long-term research project. The definitions of events for logging were provided by psychologists and correspond to the events which are normally observed during personal sessions with psychologists. In comparison with this traditional approach, our automated logging system has one considerable advantage. It removes the posibility that the conclusions made from observations are influenced by the subjective view and experience of the psychologist.

6 Conclusion

In this paper we described the novel framework enabling the integration of Kinect and Leap Motion sensors with Java Monkey Engine. The main aim of the framework was to provide the users with an easy way to develop various applications. The aplicability was demonstrated on the Labyrinth mini-game which was designed with respect to children with specific needs. As this mini-game should also serve for further evaluation of the hand-eye coordination and the ability of the player to focus, the event logging system was implemented and integrated. This helps the psychologists to measure and easily compare the improvements.

In the future, the framework could also support the recording of audio and video stream captured by Microsoft Kinect and possibly analyzing these streams to reveal specific patterns in user behavior specified by the psychologists.

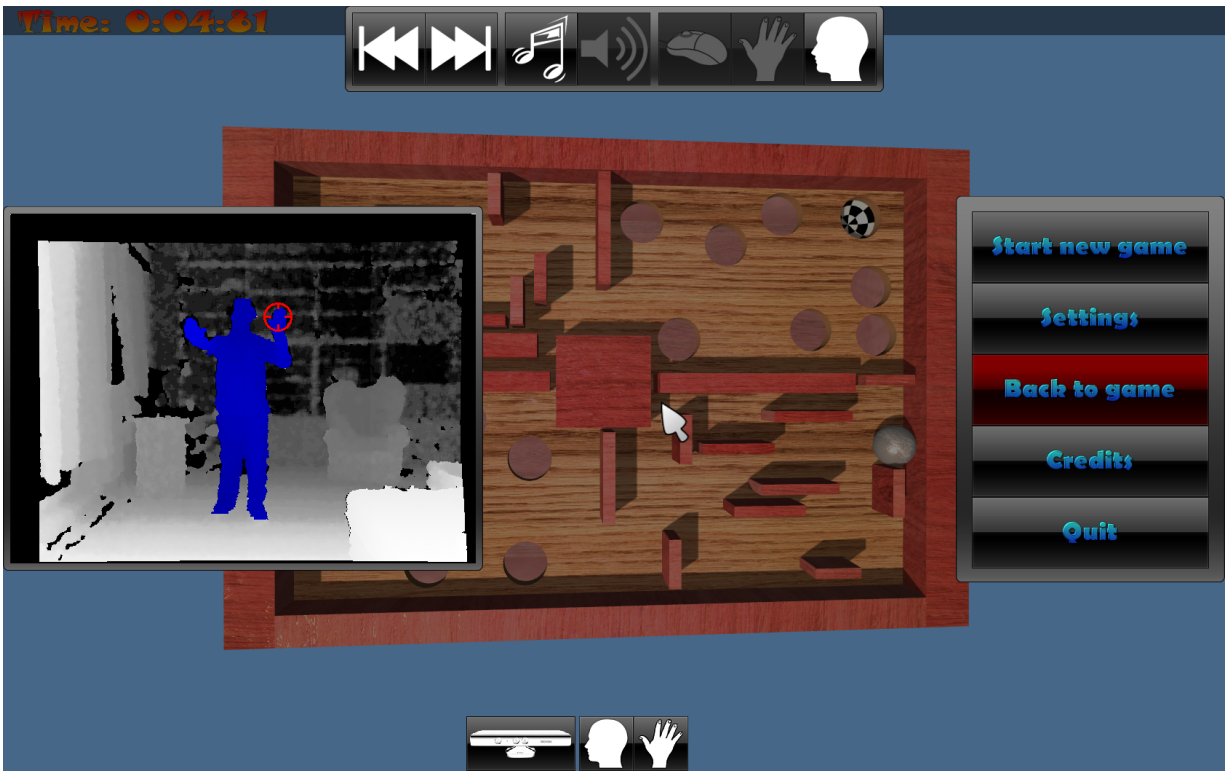


Figure 5: Application controlled by Microsoft Kinect. The depth map and GUI are displayed. The red cross represents the tracked point (right hand), the tracked person is highlighted. The left hand performs the *Arm Forwards* gesture.

References

[1] Daphne Bavelier, C. Shawn Green, Doug Hyun Han, Perry F. Renshaw, Michael M. Merzenich, and Douglas A. Gentile. Brains on video games. *Nature Reviews Neuroscience*, 12(12):763–768, 2011-11-18.

[2] Mark W. Bowyer, Kevin A. Streete, Gilbert M. Muniz, and Alan V. Liu. Immersive virtual environments for medical training. *Seminars in Colon and Rectal Surgery*, 19(2):90–97, 2008.

[3] Ann C. Childress and Sally A. Berry. Pharmacotherapy of attention-deficit hyperactivity disorder in adolescents. *Drugs*, 72(3):309–325, 2012.

[4] Erwin Coumans, et al. Bullet Engine. `http://code.google.com/p/bullet/`. [cit. 2014-03-12].

[5] Jan Flasar, Luděk Pokluda, Radek Ošlejšek, Pavel Kolčárek, and Jiří Sochor. Vrecko: Virtual reality framework. In *Theory and Practice of Computer Graphics 2005*, pages 203–208, Canterbury, 2005. Eurographics Association.

[6] Jeffrey M. Halperin, Anne-Claude V. Bédard, and Jocelyn T. Curchack-Lichtin. Preventive interventions for adhd. *Neurotherapeutics*, 9(3):531–541, 2012.

[7] Yasaman Hashemian and Marientina Gotsis. Adventurous dreaming highflying dragon. *Proceedings of the 4th Conference on Wireless Health - WH '13*, pages 1–2, 2013.

[8] Java Monkey Engine 3.0 – Java OpenGl Game Engine. [online]. `http://jmonkeyengine.org`. [cit. 2014-03-08].

[9] Dubi Lufi and Elisheva Gai. The effect of methylphenidate and placebo on eyehand coordination functioning and handwriting of children with attention deficit hyperactivity disorder. *Neurocase*, 13(5-6):334–341, 2008-05-15.

[10] Walter Matthys, Louk J. M. J. Vanderschuren, Dennis J. L. G. Schutter, and John E. Lochman. Impaired neurocognitive functions affect social learning processes in oppositional defiant disorder and conduct disorder. *Clinical Child and Family Psychology Review*, 15(3):234–246, 2012.

[11] PrimeSense: NiTE Middleware – the Natural Interaction Engine. [online]. `http://www.primesense.com/wp-content/uploads/2012/10/PrimeSense-NiTE-Middleware-A4-Lo.pdf`. [cit. 2014-03-08].

[12] Daria Nitescu. Evaluation of pointing strategies for microsoft kinect sensor device. Diploma thesis, University of Fribourg, Department of Informatics, 2012.

[13] OpenNI organization: OpenNI Programmer's guide. [online]. `http://www.openni.org/openni-programmers-guide/`. [cit. 2014-03-08].

[14] Thomas D. Parsons, Todd Bowerly, J. Galen Buckwalter, and Albert A. Rizzo. A controlled clinical comparison of attention performance in children with adhd in a virtual reality classroom compared to standard neuropsychological methods. *Child Neuropsychology*, vol. 13(issue 4):363–381, 2007-06-11.

[15] Maria T. Schultheis and Albert A. Rizzo. The application of virtual reality technology in rehabilitation. *Rehabilitation Psychology*, vol. 46(issue 3):296–311, 2001.

[16] Robert J. Stone. The (human) science of medical virtual learning environments. *Philosophical Transactions of The Royal Society B*, 366(1562):276–285, 2011.

[17] Dorothy Strickland. Virtual reality for the treatment of autism. In *In G. Riva (Ed.), Virtual reality in neuro-psycho-physiology (pp. 8186*. IOS Press, 1997.

[18] Stacy C. Wegrzyn, Doug Hearrington, Tim Martin, and Adriane B. Randolph. Brain games as a potential nonpharmaceutical alternative for the treatment of adhd. *Journal of Research on Technology in Education*, 45(2):107–130, 2012.

Color Distribution Transfer for Mixed-Reality Applications

Stefan Spelitz*
Supervised by: Reinhold Preiner†

The Institute of Computer Graphics and Algorithms - Computer Graphics Group
Vienna University of Technology
Vienna / Austria

Abstract

In mixed-reality environments it is essential to integrate virtual objects seamlessly into a real scene. Virtual objects should have similar appearances to those of real objects captured by a video camera. The aim of this work is to integrate an existing method of statistics-based color mapping into mixed-reality applications. This allows us to simulate current luminance conditions of the scene as well as changes in the camera driver settings and apply them onto virtual objects. This paper contains a fast-running approach to provide a color mapping between virtual objects and the real scene, which can be used in real-time applications. The results show that this method increases the immersion of virtual objects in a real scene.

Keywords: Mixed Reality, Augmented Reality, Color Transfer, Differential Rendering, Tone Mapping

1 Introduction

Mixed-reality attempts to embed virtual objects into the real world. This is typically done by using a video stream of a camera, merging rendered objects into the video input and presenting the result on an output device, like a PC monitor or a mobile device. One goal might be to create a perfect illusion such that virtual objects cannot be distinguished from real, existing objects. These techniques may be used in (but are not limited to) edutainment systems, architectural and urban visualizations or for marketing reasons.

To create such an illusion, different approaches already exist. Klein and Murray [4] introduced methods to simulate camera artifacts, e.g. distortion, chromatic aberrations and blur on virtual objects. These artifacts are applied onto the virtual objects, before they are merged with the video stream. Other methods [6] are taking direct and indirect illumination effects into account to simulate mutual lighting effects between real and virtual objects.

Although these methods create accurate results, they don't consider matching the actual colors between the virtual objects and the camera scene.

(a) *Source* **(b)** *Target* **(c)** *Result*

Figure 1: *Color matching between two images. Source image (as in the one whose colors were changing) on the left. Target image (as in the one whose colors we want to match to) in the middle. Result after using color transfer [12] in CIELab color space on the right.*

Cameras map the radiance of the real world to an image with RGB information, by using a camera-specific transfer function. Virtual objects which are merged with a camera image are easily categorized as artificial, because their colors don't match those in the scene (see Figure 5, 'Tonemapping' column for examples). The colors in the image typically depend on the global illumination conditions as well as the hue, saturation or white balance settings of the camera. With a stable color mapping function the behavior of the camera can be simulated and it is possible to adapt the colors of virtual objects to better suit the colors available in the camera image. Virtual representations of real objects registered within the system are desirable, but not necessary for the algorithm to work.

The work in this paper is based on Differential Instant Radiosity [6], which is used for the basis framework. My method attempts to improve the previous work of 'adaptive camera-based color mapping' [7] by using a global, statistics-based mapping, known as 'color transfer' [12] in the domain of mixed-reality.

This paper's main contributions are:

- An analysis of suitable color spaces in the domain of statistics-based color matching (Section 3)
- A novel approach of using 'color transfer' [12] to globally adapt the colors of virtual objects to that of a camera image (Section 6)
- A fast-running implementation of the method as GPU shader code (Section 6.1)

*stefan.spelitz@tuwien.ac.at
†preiner@cg.tuwien.ac.at

2 Related Work

This paper is based on my bachelor thesis [15] which contains additional information and an additional color mapping function. In the remainder of this section, the related work on histogram matching between two images and existing applications in augmented reality are discussed.

Statistics and Histogram Matching: The first popular method by Reinhard et al. [12] matches the mean and standard deviation of a source image to that of a target image. This is done separately for each color channel in L$\alpha\beta$ color space. Based on this method Kim et al. [3] proposed a method which also works in L$\alpha\beta$ color space but is using an additional pre-processing of the source image's colors and transforms only the α and β channels. The method of Reinhard et al. produces convincing results, but it works with statistical data of the whole image and thus can create new colors in the result by mixing up two or more colors of the target image. Xiao and Ma [17] tried to solve this problem with histogram matching and a post-processing step to preserve the gradients of the source image.

Another way of performing histogram matching is to create an image dependent color space instead of using a fixed one. This is done by eliminating the coherence between the color channels, also with the idea to perform color mapping on each color channel separately. A non-linear mapping with this approach has been proposed by Grundland and Dodgson [2]. Similarly, not depending on a fixed decorrelated color space, Xiao and Ma [16] decompose the source and target image data into its principal components (with singular value decomposition) to perform a one-dimensional color mapping.

Besides these methods which perform color mapping on each color channel, there are those which are trying to solve the color mapping in N-dimensions. Neumann and Neumann [8] are using a computationally simple, permissive, or optionally strict 3D histogram matching. Instead of using opponent color channels they are using a cylindric color space to map hue, lightness and saturation as their main attributes. Another N-dimensional mapping has been proposed by Pitié et al. [10]. In their work they use a N-dimensional probability density function transformation with an involved post-processing step, which matches the gradient field of the output image to the source image.

The requirements of a color mapping function, for usage in real-time mixed-reality applications are to transfer the colors without additional user interaction and to allow real-time framerates. Methods which allow the user to control the amount of transformation (like [11]) are useful for a manual matching of arbitrary images, but in real-time applications simple, fast-running methods are necessary, which must be well suited for generic automated tasks.

Color Matching in Augmented Reality: Knecht et al. [7] proposed an algorithm to match the colors between virtual objects and the camera scene. This is done by creating color sample pairs based on matching similar colors as well as through a heuristic function. A color mapping function is then derived from the color sample pairs. The method performs well if there are similar colors on virtual and real objects. If there is not enough matching information in the camera scene or the differences between the colors of virtual and real objects are too extreme, a correct mapping will likely fail and lead to incorrect colors in the final result.

In the recent work of Oskam et al. [9], they provided a color balancing technique while tracking a marker in the real scene. They use the marker to compare its colors with a virtual representation and build up correspondences. To find corresponding points they use a RANSAC-based algorithm. A radial basis function interpolation is used to propagate the correspondences to the remaining color space. The algorithm creates plausible results, but needs initial correspondences.

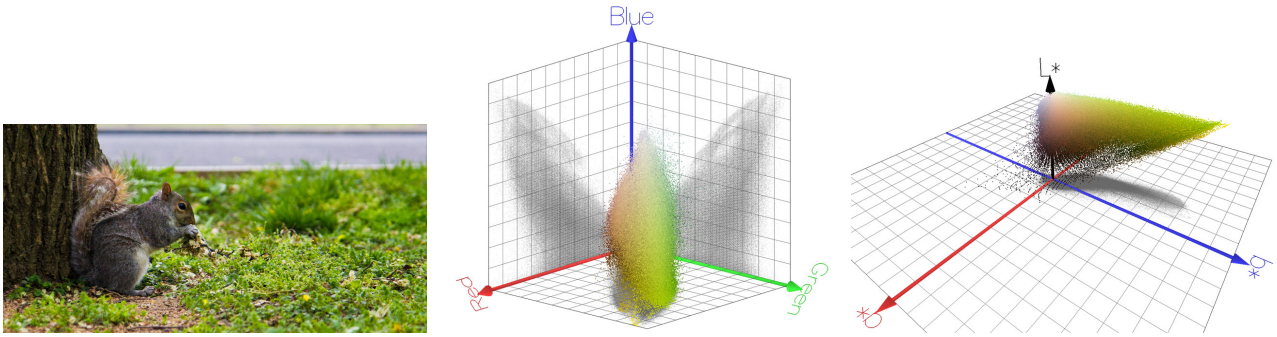
3 Color Spaces

As stated in the previous section, color mapping algorithms can typically be categorized into two classes. There are those which operate in a N-dimensional color space and those which operate on each color axis separately. The mapping algorithm presented in this paper is working on a per-color-axis basis. It has been observed that if the channels can be made strongly decorrelated then image processing can be done in each channel independently [12]. So it is assumed, that the choice of the color space is important for algorithms which perform one-dimensional matching.

The RGB color space tends to have axes, which are strongly correlated. An example is displayed in Figure 2. In this example, the values are typically small in each channel for dark colors. The values are getting larger as the luminance rises. If the blue channel's values are large, then most values in the red and green channels are getting larger, too. This results in an almost diagonal distribution between the axes, which signals a strong correlation. Therefore, when changing the color of a pixel to match another one, it is necessary to change all color channels simultaneously. This results in more complex color matching techniques. Thus, for this paper, the RGB color space will not be used to perform color mapping.

CIELab (also CIE L*a*b*) is a device independent color space with three axes. 'L' represents the lightness of the color with a range from 0 (black) to 100 (white). The other two axes are representing the blue-yellow (channel 'b') and red-green (channel 'a') chromatic opponent channels with an unbounded range. It is a non-linear transformation of the CIE XYZ color space, while still remaining reversible. It is considered to be perceptually uniform. This means, that the euclidean distance of two colors in CIELab are reflected as equally distant in perception.

Reinhard and Pouli [13] compared the quality of color mapping in the domain of different color spaces (e.g. CIELab, L$\alpha\beta$, HSV, XYZ) in combination with several



(a) *Input image* **(b)** *RGB correlation* **(c)** *CIELab correlation*

Figure 2: *Decorrelation properties of color spaces. Color distribution of image (a) is plotted in RGB space (b) and in CIELab color space (c). RGB shows an almost diagonal distribution on each pair of axes. CIELab distribution is along L* and b* axes. Plots created with ColorSpace software [1].*

environment settings (e.g. indoor, day, night). They concluded in their work:

> *'Surprisingly, we find that CIELAB, if used with illuminant E as the white point leads on average to the best performance, yielding a plausible colour transfer in 77 % of all cases tested.'*

Although it seems plausible to see more indoor-specific mixed-reality applications, color mapping in mixed-reality environments cannot make assumptions about the environment it is used in. Therefore it is necessary to choose a color space with overall good performance results. Because CIELab (E) performs well in all tested environments and especially in indoor areas, it is the color space of choice in this paper.

4 Differential Rendering

Knecht et al. [6] developed a method called 'Differential Instant Radiosity' (DIR), which is the core of the framework used in this paper. They combine differential rendering (DR) and instant radiosity to be used in mixed-reality applications. By doing so it is possible to calculate effects like shadow casting and indirect illumination between real and virtual objects. The main aspect used from this paper is the work about differential rendering.

To use DR the following information is needed:

- The camera image (CI)
- One global illumination (GI) solution for the local scene containing virtual and real objects (LS_{rv})
- One GI solution containing only the geometric representation of real objects (LS_r)

Illumination is captured using a fish eye camera to adapt the scene to environment lighting. For details on how the global illumination solutions are obtained, see [6]. The actual DR process is done by creating the difference between LS_{rv} and LS_r after both solutions have been tone and color mapped. The difference (i.e. $LS_{rv} - LS_v$) is then applied to a masked CI to obtain the final result.

By using a virtual representation of real objects (i.e. LS_r) it is possible to measure the difference between LS_r and the CI. This measurement can then be used to do the actual mapping between virtual objects and the real scene.

It is assumed that geometric representations of real world objects (at least some objects) are available. The LS_r and LS_{rv} solutions contain geometric models of high dynamic range (HDR). Because the captured camera image is only in low dynamic range (LDR) a tone mapping operation is necessary. The chosen global tone mapping operator is based on the work of Reinhard et al. [14]. After the tone mapping has been applied, all information is in a common LDR color space.

As a result the method creates a merged image which consists of the camera image, virtual objects, shadows and reflections.

5 Application Flow

Figure 3 shows the abstract application flow with the help of an example. In the camera image four real existing objects are available. The wooden surface and the color chart have similar geometric representations in the application. The red figure and the book have no virtual representation. Therefore LS_r contains the wooden surface and the virtual representation of the color chart. LS_{rv} contains in addition to the content of LS_r, the object to be rendered into the real scene, which is another color chart. The actual color mapping process is divided into four stages, which will be explained next.

Stage 1: After LS_r and LS_{rv} have been tone mapped, they are converted together with the camera image to the CIELab color space. This is done to minimize correlation between the color axes, so that manipulations of one color axis don't affect the other axes as well.

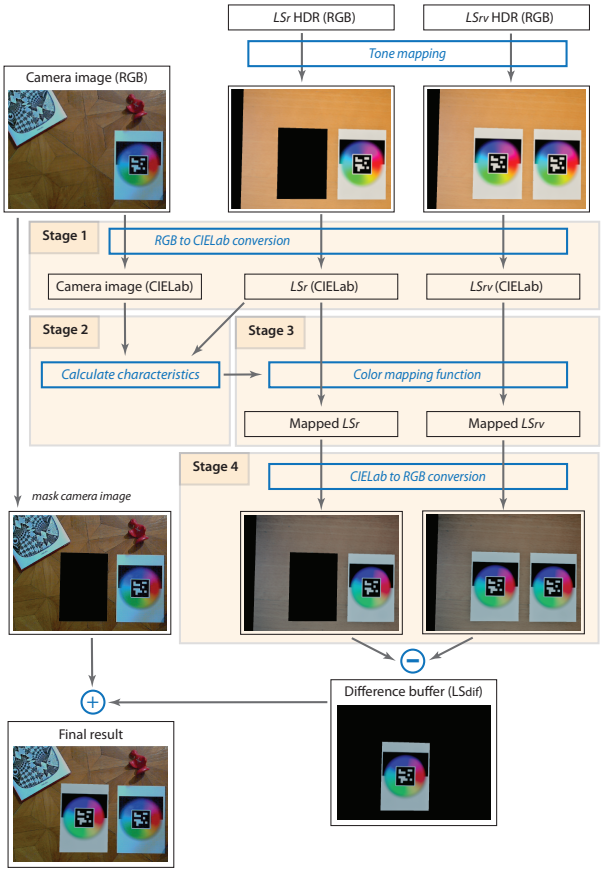


Figure 3: *The application workflow. By calculating the characteristics of the real-world in comparison to LS_r, the color mapping function is applied to the LS_r and LS_{rv} solutions. The color mapping operates in the decorrelated CIELab color space. The difference between the buffers is then merged with the camera image to create the final result.*

Stage 2: For the actual color mapping function we need to calculate the differences between the representation of the real-world (i.e. LS_r) and the actual real-world itself (i.e. the camera image). By determining the characteristics of these two images, the resulting values can be used by the 'color mapping function' (CMF).

Stage 3: The CMF applies the characteristics to both, LS_{rv} and LS_r, in order to convert their colors to match those of the camera image. This will be explained in detail in Section 6.

Stage 4: The CIELab conversion was only necessary to calculate the characteristics and perform the color mapping. So after the color mapping is done, LS_{rv} and LS_r will be converted back to RGB color space.

Obtaining the result: We then get the final result by calculating the difference

$$LS_{dif} = LS_{rv} - LS_r$$

and adding the difference buffer to a masked version of the camera image.

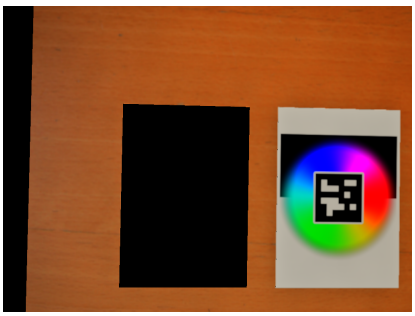


Figure 4: *This figure shows the LS_r solution with two virtual representations of real objects (table, color-chart). The black area on the left indicates the end of the virtual table. The black area in the center is the place for the virtual object to be rendered.*

6 Color Transfer

The color mapping function is based on the work of Reinhard et al. [12]. Although its primary purpose is to transfer the colors between two images (see Figure 1), in this paper three images will be involved.

The first step is to calculate the color characteristics of the source (i.e. LS_r) and the target (i.e. CI) images. The characteristics are the mean and the standard deviation of the respective color distributions. Denoted by μ_s, μ_t and σ_s, σ_t. The next step is to convert each data point (x_i) of the LS_r and LS_{rv} solutions:

$$x_i^* = (x_i - \mu_s)\frac{\sigma_t}{\sigma_s} + \mu_t$$

So we move the data points by the source mean, scale them by using the standard deviations and move them again by adding the target mean. Please note, that the same transformation is applied to LS_{rv} and to LS_r. Therefore colors which are the same in both solutions will remain equal after the mapping. This is an important feature, necessary for differential rendering.

6.1 Color Characteristics on the GPU

One aspect when using the color transfer method is to calculate the color characteristics in an efficient and fast way. We will concentrate on doing that in the following section with respect to GPU shader considerations.

Calculating the mean: The arithmetic mean is defined as:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

where x_i is a data point of an image with dimensions $[w,h]$. With n being the number of data points, which is $n = w \cdot h$.

Calculating the arithmetic mean is easily done by creating mipmaps. Mipmapping will typically only work on quadratic textures. Creating a quadratic texture from a

rectangular one is done by bilinear point sampling. Because of using linear interpolation the mean does not get affected.

The mean must be calculated for the camera image as well as for LS_r. Although calculating the mean for the CI is straight forward, this is not the case for LS_r. It contains some areas with no information (see Figure 4). These black areas must not have any influence on the calculated mean value.

The count of pixels in the black areas is denoted as n_{zero}. The mean (μ), obtained by the mipmapping operation, can be modified to exclude the black areas by using:

$$\mu_{correct} = \frac{\mu \cdot n}{n - n_{zero}}$$

Note 1: This works only because the data points we want to exclude from the mean calculation have a value of zero.

Note 2: The corrected mean calculation could have been applied to the masked version of the camera image, too. The masked version contains black areas at each point where a virtual object will be placed at. Although it is possible, there are some drawbacks. The black areas are 'lost information'. These areas won't be included in the calculation, so we have less information about the target environment we want to map to. Therefore we lose precision in the color mapping. In addition, if a virtual object covers the whole scene (and $n_{zero} = n$), there won't be any information available from the camera image and thus the mapping would fail.

Calculating the standard deviation: Because the standard deviation is the square root of the variance, we will concentrate on calculating the variance. The variance for discrete values is defined as:

$$var = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \tag{1}$$

The variance can easily be calculated. First, calculate the squared deviation from the mean for each data point in the texture (i.e. $(x_i - \mu)^2$). This operation can be executed in one shader pass. The next step is to make the texture quadratic and execute mipmapping to get the arithmetic mean of the sum of squared deviations. The result of mipmapping is the variance.

When calculating the variance a similar problem occurs as when calculating the mean for LS_r. Because some data points (count is n_{zero}) shall be excluded from the calculation, we need to correct the variance calculation. Excluded data points have values of zero, so we can rewrite the variance Eq. 1 to be:

$$var = \frac{1}{n} \cdot (n_{zero} \cdot (0 - \mu)^2 + \sum_{i \in R}(x_i - \mu)^2) \tag{2}$$

with R being the remaining set of data points and having $|R| = n - n_{zero}$. The corrected variance only contains the remaining data points and is therefore, according to Eq. 1:

$$var_{correct} = \frac{1}{|R|} \cdot \sum_{i \in R}(x_i - \mu)^2 \tag{3}$$

Which is equal to (using Eq. 2):

$$var_{correct} = \frac{1}{|R|} \cdot (var \cdot n - n_{zero} \cdot \mu^2) \tag{4}$$

This shows that it is possible to exclude zero valued data points by using the mipmap-calculated variance and applying Eq. 4 to get the corrected variance.

7 Results

The PC used for the test results has an Intel Core i7-950 Quad 3.06 GHz CPU with 6 GB RAM and a nVIDIA GeForce 9800 GTX+ graphics card. The operating system was a Microsoft Windows 7, 64 bit. The framework was developed in C# using the DirectX 10 API in conjunction with the SlimDX library. The used shader language was HLSL.

To see the capabilities of the 'color transfer' method, a test setup was created and evaluated under different contrast and saturation settings in the camera driver.

As seen in Figure 5(a), the test setup contains multiple real-existing objects. A wooden surface, a book, a red figure and a color chart to the right. The application has only two registered virtual objects, which is the wooden surface and the color chart.

The goal is to render a virtual color chart object (placed to the left of the real-existing one) which matches the color settings of the surrounding environment. The virtual object representation should match the real object's appearance. If there is no real object for comparison available, the virtual object should fit into the environment in a harmonic way without losing its overall color appearance.

Broadly speaking there are three different environments, in which the methods operate. These are

- the 'default state' without tweaks of the camera driver settings (Figure 5(a))
- the scene with changed camera driver settings (i.e. contrast, saturation) (Figure 5(b)-(e))
- the scene with obstacles occluding the color chart (Figure 5(f)-(g))

By occluding the real-existing color chart with obstacles it is impossible to find a direct mapping between the colors of the virtual and the real color chart. The color mapping is still expected to deliver good results even if there is no real object for a virtual representation available in the scene.

7.1 Comparison

The following methods have been compared:

- Color transfer (see Section 6)
- Adaptive camera-based color mapping by Knecht et al. [7]
- Photographic tone reproduction (tone mapping) by Reinhard et al. [14]

The tone mapping operation by Reinhard et al. is influenced by the global illumination solution and does not react on changes in the scene or camera settings. It is a tone mapping operator and not a color mapping function. In Figure 5 it is used as a comparison of how the virtual object would look like without any color adaptation.

The method of Knecht et al. is based on a heuristic which creates color sample pairs. These pairs are used to define a color mapping function. The heuristic works well as long as there is a virtual and a real representation of the same object in the scene (Figure 5(a)-(e)). It fails to find a suitable color mapping with the real-existing color checker board (Gretagmacbeth - ColorChecker Digital SG) occluding the color chart (Figure 5(f)). In Figure 5(g) with only some real-existing colored paper spread out, the method nearly completely adapts to the existing colors in the scene, which typically is not the desired result. Compared to the tone mapping operation, it is a reasonably fast color mapping technique.

Using the 'color transfer' method results in a good adaptation to changed camera settings (Figure 5(b)-(e)), but some color intensity is lost in the yellow, green and cyan areas of the virtual object. Especially when using a high contrast level (Figure 5(b)) or the color checker board (Figure 5(f)). In the scene with only some colored paper spread out (Figure 5(g)), the method does not adapt to the new colored environment but only attempts to darken the colors, when compared to the 'tonemapping' operator. The performance results are quite similar to the results of Knecht et al.

Essentially, the conclusion of the tests is that the 'color transfer' method is the better choice for color mapping in mixed-reality applications.

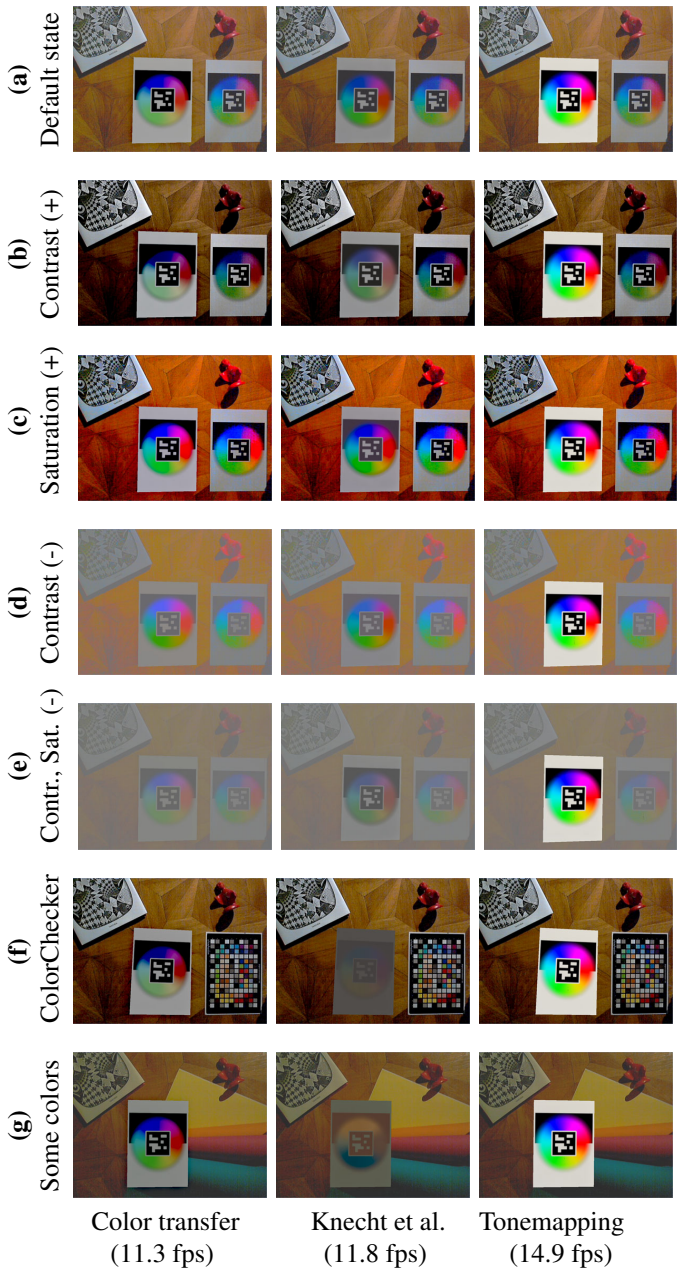


Figure 5: *Comparison of different algorithms with different settings and environments.*

7.2 User Study

The 'color transfer' method, presented in this paper, has been evaluated by Knecht [5] as part of a web survey. In this evaluation the participants were able to choose between two mixed-reality scenes in each trial. The task was to choose the image where the virtual objects fit the scene subjectively better than in the other image. Overall there were 65 trials available to the users. The survey tested multiple combinations of different rendering modes (global illumination, color mapping, camera artifacts).

It showed that the 'color transfer' overall has a negative impact on perceived quality. As a possible explanation, Knecht concluded that the color mapping changes the saturation levels of the virtual objects in a way noticeable to the user. All test images were used with standard camera settings. Therefore the possible strength of the algorithm of adapting to highly changed camera settings (e.g. saturation, contrast) was not tested.

8 Conclusions, Limitations and Future Work

Existing methods, known from computational photography, for transferring the color distribution from one image to another have been combined with differential rendering to a novel approach, usable in the field of mixed-reality applications. A color mapping technique has been presented, which dynamically adapts in each rendered frame to the internal changes of the camera settings. By using this method, colors of virtual objects are closely related to the colors of the camera image, which results in a better immersion of virtual impressions in a real scene.

It has been shown that the presented 'color transfer' mapping algorithm is superior to existing approaches. Furthermore by combining this color mapping with the simulation of camera artifacts [4] (like distortion and blur) a high quality illusion could be created, resulting in virtual objects, which may be undistinguishable from real ones.

8.1 Limitations

Because of using differential rendering, the method presented in this paper needs virtual representations of real object's geometry. This is necessary to determine the differences between the representation and the real scene captured by the camera. These differences are then applied by using the color mapping function onto the virtual objects. Although the algorithms also work without a virtual representation of the environment, the results are less precise because of the lack of mapping information. Therefore this approach should only be used in mixed-reality systems which support the representation of the real scene.

Another obvious limitation is the color mapping function. This function is working with statistical data of the whole scene and tries to adapt colors of virtual objects to the color average of the scene. This doesn't need to be correct in every possible scenario. Especially if there are multiple areas in the scene with huge differences in luminance or color setting, the average of the scene might not be the correct mapping target. A possible solution to this would be to divide the scene into sections and perform a color mapping for each section.

As with other statistics and histogram matching methods, the 'color transfer' algorithm is good for automatic mapping. On the other hand it does not provide direct control over the color mapping process and thus may not be suited for any situation.

8.2 Future Work

The user study of Knecht [5] showed that the overall impression of color-mapped virtual objects is worse than without color mapping when using standard camera settings. Knecht also stated that a separate study should be performed, investigating the impact of changed camera settings on the perceived quality of color-mapped mixed-reality scenes. Nonetheless, the survey showed that there is the need for improvement for the default case, when the standard camera settings are used.

References

[1] Philippe Colantoni. Colorspace software. http://www.couleur.org. Accessed: 2014-03-15.

[2] Mark Grundland, Neil Dodgson, Reiner Eschbach, and Gabriel Marcu. Color histogram specification by histogram warping. *Color Imaging X: Processing, Hardcopy, and Applications*, 5667(1):610–621, 2005.

[3] Jae Hyup Kim, Do Kyung Shin, and Young Shik Moon. Color transfer in images based on separation of chromatic and achromatic colors, 2009.

[4] Georg Klein and David Murray. Compositing for small cameras. In *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*, ISMAR '08, pages 57–60, Washington, DC, USA, 2008. IEEE Computer Society.

[5] Martin Knecht. *Reciprocal Shading for Mixed Reality*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, December 2013.

[6] Martin Knecht, Christoph Traxler, Oliver Mattausch, Werner Purgathofer, and Michael Wimmer. Differential instant radiosity for mixed reality. In *Proceedings of the 2010 IEEE International Symposium on Mixed and Augmented Reality (ISMAR 2010)*, pages 99–107, October 2010.

[7] Martin Knecht, Christoph Traxler, Werner Purgathofer, and Michael Wimmer. Adaptive camera-based color mapping for mixed-reality applications. In *Proceedings of the 2011 IEEE International Symposium on Mixed and Augmented Reality (ISMAR 2011)*, pages 165–168. IEEE/IET Electronic Library (IEL), IEEE-Wiley eBooks Library, VDE VERLAG Conference Proceedings, October 2011. E-ISBN: 978-1-4577-2184-7.

[8] Laszlo Neumann and Attila Neumann. Color style transfer techniques using hue, lightness and saturation histogram matching. In *Computational Aesthetics in Graphics, Visualization and Imaging 2005*, pages 111–122, 5 2005.

[9] Thomas Oskam, Alexander Hornung, Robert W. Sumner, and Markus Gross. Fast and stable color balancing for images and augmented reality. In *3D Imaging, Modeling, Processing, Visualization*

and Transmission (3DIMPVT), 2012 Second International Conference on, pages 49–56, Oct 2012.

[10] François Pitié, Anil C. Kokaram, and Rozenn Dahyot. Automated colour grading using colour distribution transfer. *Computer Vision and Image Understanding*, 107(1–2):123 – 137, 2007.

[11] Tania Pouli and Erik Reinhard. Progressive color transfer for images of arbitrary dynamic range. *Computers & Graphics*, 35(1):67 – 80, 2011.

[12] Erik Reinhard, Michael Adhikhmin, Bruce Gooch, and Peter Shirley. Color transfer between images. *Computer Graphics and Applications, IEEE*, 21(5):34–41, 2001.

[13] Erik Reinhard and Tania Pouli. Colour spaces for colour transfer. In *Proceedings of the Third international conference on Computational color imaging*, CCIW'11, pages 1–15, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *IN: PROC. OF SIGGRAPH'02*, pages 267–276, 2002.

[15] Stefan Spelitz. Color distribution transfer for mixed-reality applications. Bachelor thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, October 2012.

[16] Xuezhong Xiao and Lizhuang Ma. Color transfer in correlated color space. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, VRCIA '06, pages 305–309, New York, NY, USA, 2006. ACM.

[17] Xuezhong Xiao and Lizhuang Ma. Gradient-preserving color transfer. *Computer Graphics Forum*, 28(7):1879–1886, October 2009.

Multi-frame Rate Augmented Reality

Philipp Grasmug*
Supervised by: Dieter Schmalstieg†

Institute of Computer Graphics and Vision
Graz University of Technology
Austria

Abstract

In this work we present a method for improving the visual quality of an augmented reality system. By combining the characteristics of two different sensors, we increase the spatial resolution of a video stream using sub-pixel accurate image registration. Using the optical flow to solve the correspondence problem, we can estimate the depth of the scene, which is further used to compute immersive augmented reality effects. By decoupling the rendering process of the augmented information from the displaying frequency of the system, we can augment the scene using computationally expensive rendering techniques. We utilize image-based rendering to overcome the resulting temporal artifacts. Finally, we evaluated our methods by comparing the achieved quality with conventional augmented reality methods.

Keywords: augmented reality, interactive superresolution, image registration, workload distribution

1 Introduction

Increasing the spatial resolution of images is normally achieved by developing new sensor chips for cameras with a higher pixel density or larger sensors. Both of these improvements have drawbacks in terms of image quality (smaller pixels mean less light, leading to more noise) and efficiency. Also, capturing images at a high resolution (HR) means that a lot of data has to be transferred, which requires a bus with sufficient transfer rate.Alternatively one can combine information from multiple subsequent images of the current scene or arbitrary images from a database to compute a realistic or at least plausible high resolution version of the input image. We combine data from two sensors captures at different temporal and spatial resolution to create a true high resolution output. The hardware setup of our system looks like this: The first sensor provides a video stream in LR, but at a high frame rate (e.g., 30 Hz). A second sensor, which is placed right next to the first one, captures still images in HR at a slow frame rate. The latest high resolution image is registered to the current low resolution video frame to compute the desired output.

The information visualized in augmented reality (AR) applications ranges from simple textual information to computationally expensive visualizations, which can not be computed in real-time. If a high resolution is required. By combining images from different sensors or from a single sensor captured in different spatial resolutions, we can achieve interactive super-resolution of the input video stream. To match the high resolution of the input, we present a strategy for distribution the workload of computationally expensive rendering tasks over several frames using image-based rendering to suppress temporal artifacts. For both tasks we utilize the GPU to speed up the computation. The image registration needed for our super resolution approach can be computed highly efficient on programmable graphics hardware while the use of a GPU for rendering the augmentations is self-explanatory. Figure 1 shows a overview of the main parts of our system which are described in detail in the following sections.

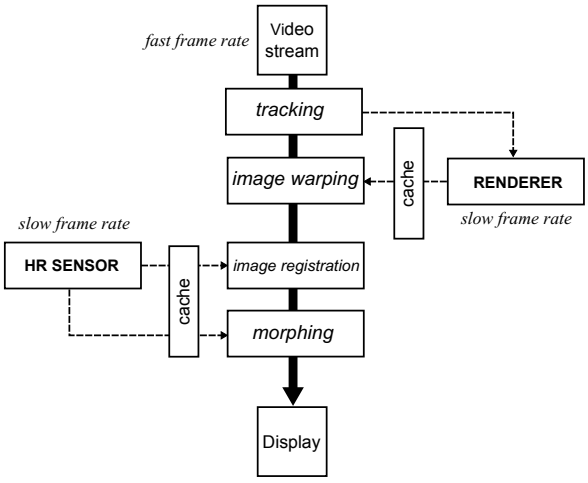


Figure 1: Overview of the main parts of our system.

*grasmug@icg.tugraz.at
†schmalstieg@icg.tugraz.at

2 Previous Work

Improving images in terms of spatial resolution is relevant in many fields and applications. Super-resolution (SR) is an algorithmic approach that combines information from several images or from large image databases into a true or at least a plausible high resolution version of the low resolution (LR) source. Different approaches in the spatial and the frequency domain have been proposed.

Algorithms in the frequncy domain exploit the aliasing of LR images to reconstruct a HR image. Huang and Tsai [10] presented an algorithm that built upon the relative motion between the LR images. The method utilizes the aliasing relationship between the continuous Fourier transformation of the HR image and the discrete Fourier transformation of the LR images.

Example-based techniques (also called image hallucination) [6, 5, 1] try to model the relationship between LR and HR images using corresponding patches of LR and HR images that do not necessarily show the same scene. The correspondence between those pairs is learned from a large database and used to synthesize a plausible HR image. Bhat et al. [3] proposed a system similar to ours that uses static images to enhance a video of the same scene. The method computes view-dependent depth data for the images and the video and uses this information to apply a variety of effects including super resolution as an offline post-processing step. Known SR algorithms rely on optimization and/or machine learning which makes them slow and not usable in interactive applications like augmented reality. Our algorithm in contrast is based on optical flow computation and allows for interactive use.

Render caching reuses information from previous frames to speed up the computation of the current one. When the frame rate is high, the changes from a previous frame to the current one are small and therefore, the temporal coherence is high. This information can then directly (e.g., color of a pixel) or indirectly (e.g., intermediate results) be reused.

The term *render cache* was introduced by Walter et al. [17], who used it as a data structure to speed up rendering of otherwise none interactive methods. Yu et al. [18] proposed a GPU implementation of the forward reprojection algorithm, which uses a per pixel disparity vector to compute the new position. Implementing this approach is difficult and can be computationally expensive. Didyk et al. [4] proposed an efficient method, which fits a coarse, regular mesh grid to the cached image. The mesh is aligned to the depth discontinuities of the cache. Each vertex is warped to its new position using the cached image as texture. Holes are avoided by stretching of the mesh and visibility is resolved by fold overs.

The applications for render caching are numerous. Sitthi-Amorn et al. [16] proposed an algorithm for the acceleration of pixel shader computations by directly reusing information from the cache whenever available. A relevant work on spatio-temporal upsampling of renderings was published by Herzog et al. [9]. By combining multiple low resolution renderings, using a modified joint-bilateral filter, a high-resolution image can efficiently be computed. Render caching focuses on reuse of results from previous frames for rasterization based techniques while our workload distribution scheme targets oversampling based methods and exploits their properties. This allows different approaches which cannot directly be applied to rasterization.

Image-based rendering The general idea of image based rendering (IBR) is to derive a novel view from real or synthetic images. The usual way of rendering an image of a synthetic scene is by using one of the standard algorithms like, for example, rasterization, ray tracing or path tracing. Image warping [13, 14] relies on associated per-pixel depth information. This information together with the position and orientation of the camera is used to reproject each pixel into three dimensional space and then into the view of the new desired virtual camera. For synthetic scenes, this is straight forward, since storing per-pixel depth values and camera positions can be easily done during rendering. IBR is utilized in our work to compensate temporal artifacts. We further employ a method, again closely related to oversampling based rendering approaches, to address IBR related issues like resampling and disocclusion artifacts.

3 Upsampling

In this section, we describe our upsamling strategy, which is inspired by previous work in the field of SR. We present an approach that combines HR images and an LR video stream from one ore multiple sensors to compute a SR version of the LR input online. At the point in time at which the still frame is captured, both sensors show a nearly identical images of the scene (with slight differences due to the spatial placement). Over time the images begin to diverge due to the motion of the camera or the scene. To be able to use the information from the high resolution image, we have to correct this divergence. We do this by computing the optical flow between a subsampled version of the latest high resolution image and the current frame of the video stream. The resulting sub-pixel accurate displacement map is used to transform the HR image to match the current video frame. Our algorithm can be outlined by the following steps, where I_{LR} denotes the current video frame and I_{HR} the latest HR image.

1. Compute the optical flow from I_{LR} to I_{HR} and vice versa. The computation is done at the resolution of I_{LR}.

2. Compute the confidence for every pixel.

3. Upscale the flow field using bilinear interpolation.

4. For each pixel, lookup the color of I_{HR} using the computed flow field from I_{LR} to I_{HR}.

5. Blend I_{HR} with I_{LR} image according to the confidence map.

The result of the optical flow computation is crucial for the quality of the SR and mainly depends on the disparity between the pair of images, which is influenced by spatial placement of the sensors, camera motion and motion in the scene. Another associated problem is occlusion and disocclusion of objects in the scene. Due to the change in viewpoints over time, areas of the scene get revealed which are not visible in the latest HR image. This is obviously an issue for the image registration step, especially if the scene has a high depth complexity since objects are more likely to occlude each other. Therefore, we have to expect a certain error in the optical flow computation.

3.1 Confidence Map

Figure 2: Visualization of the confidence map. Red shading denote regions with low confidence like, for example, the area around the bottle, which has been disoccluded due to the movement of the camera.

In order to be able to compute an artifact-free upscaled version of the video stream, we have to account for the mentioned problems. In detail, this means that we have to detect occlusions and regions, where the computation of the flow field yields erroneous results. Common metrics like *endpoint difference* [15] and *angular difference* [2] are not suitable for our case, since they are not reference free. To establish this property, we modeled our metric based on a simple observation. If we compute the optical flow from image I_1 to I_2 and vice versa, the flow vectors should approximately be the same with inverted sign.

$$uv_{fw}(p) \approx -uv_{bw}(p + uv_{fw}(p)) \tag{1}$$

Equation 1 formulates this observation, where uv is the 2-component flow vector with respect to the image position $p = (x, y)$. Based on this equation, we derived a metric which tells us whether the flow vector at a certain position is correct or not.

$$conf = 1 - \lambda\left(\frac{|uv_{fw}(p) + uv_{bw}(p + uv_{fw}(p))|}{\alpha}\right) \tag{2}$$

$$\lambda(x) = \begin{cases} x & \text{if } x \leq 1 \\ 0 & \text{else} \end{cases} \tag{3}$$

We call this metric *confidence* and compute a map for the whole flow field (Figure 2) as defined in Equation 2, where α is a weighting factor that defines how strongly the difference is penalized. The fiducial marker in the shown image is later used to augment the scene but not for the SR approach. The formula is based on the distance between the two flow vectors interpreted as points and ideally is zero. The confidence map is recomputed every frame. This also means that we not only have to compute the flow once, but twice each frame, to be able to compute this quality metric.

To finally create the HR output stream we blend the LR image with the warped HR image weighted by the confidence map. Since the flow field is usually good in high frequency regions of the image, we can preserve those important details. Low frequency regions tend to yield worse results in terms of optical flow computation. Using information from the LR frame in those areas means only a negligible loss of information. In the case of disocclusion, no information is available for this region, which means blending with the LR image is the only meaningful solution.

3.2 Depth Estimation

With the camera extrinsics given from the tracking (fiducial markers in our case) and the intrinsics from the calibration we further need correspondences to be able to reconstruct depth by triangulation. Using the optical flow, we can compute pixelwise point correspondences. Having all this information at hand, we now can calculate a depth estimation of the scene. Using the current video frame and an older video frame with a certain delay as key frames for the estimation, we can compute the depth with the algorithm outlined in the following:

1. Compute a ray from the center of projection into the scene for both views. This ray is given by $w = c + \lambda Q^{-1} m$, where $m = [u, v, 1]^T$ is a point on the image plane and Q is defined in equation 4. The center of projection is given by $c = -Q^{-1}\tilde{q}$.

$$P = A[R|t] = \left[\begin{array}{c|c} \mathbf{q_1} & q_{14} \\ \mathbf{q_2} & q_{24} \\ \mathbf{q_3} & q_{34} \end{array}\right] = [Q|\tilde{q}] \tag{4}$$

   P_1, P_2 as well as c_1, c_2 are obtained from associated extrinsics $[R|t]$ and intrinsics A of the chosen key frames.

2. Intersect the rays. Since rays usually do not intersect at a point in $\mathscr{R}^3$, we need to computed the shortest

line between the rays. If the length of that line is below a certain threshold, it is treated as intersection.

3. From the intersection point (or the starting point of the shortest line between both rays), we can retrieve the reconstructed depth of the scene in that particular point.

Figure 3: The top image shows the computed depth map for the reference image below.

The selection of the pair of input images is essential for the depth estimation. To obtain good results, the images must have a appropriate stereo baseline. The result further depends on the correctness of the found correspondences. We need to compute the optical flow between the latest HR image and the current LR image every frame for our super-resolution algorithm. Therefore, we already have correspondences, which could be used for the depth information. Whenever the HR image is refreshed, it is nearly the same as the current image. In this case, the stereo baseline vanishes, and we cannot use it for depth estimation. Therefore, we cache the latest N frames of the LR video stream and use one out of it for the stereo matching. The image from the cache is selected either with a fixed frame distance or using an angular threshold. For both cases this method fails, if the camera movement stops. To resolve this issue, a keyframe based approach should be used in the future. Figure 3 shows the resulting dense depth map. While our SR approach is applicable to dynamic scenes too, the depth estimation is limited to static scenes.

4 Workload Distribution

In this section, we describe a workload distribution scheme that allows to compute expensive effects in real time by decoupling the rendering process. In contrast to render caching we focus on oversampling based approaches like path tracing which allows us to employ a different strategy.

To decouple the rendering process from the displaying frequency of the system, we discuss two strategies: First the computation can be *time sliced*. Only a subset of all samples per pixel is computed each frame. The final image is available after a number of frames depending on the size of the subset. A similar technique is to *spatially slice* the image by computing a sub-region of the rendering each frame with the full number of samples. Since the objects in the scene are most likely not distributed uniformly, this method can result in an unsteady frame rate. With both strategies, the final image is available after a distinct number of frames depending on the splitting criterion and on the desired total number of samples. Figure 4 illustrates both strategies.

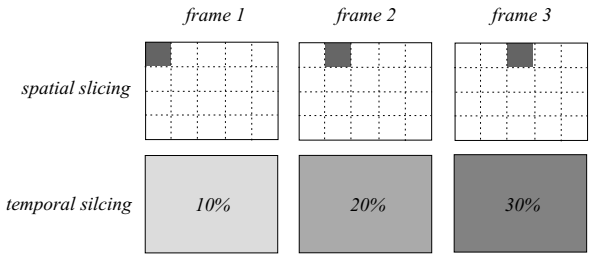


Figure 4: Spatial versus temporal slicing. In the upper row, the shaded rectangle denotes the region which is computed at frame n. In the lower row, the shading denotes the percentages of samples per pixel which have been computed to this point.

For our experiments, we implemented a path tracer to augment the scene. This algorithm is just an example of the rendering methods that can be used. Since image-based rendering, in our case image warping, only relies on per-pixel color and depth information, essentially any rendering technique can be used. The workload distribution strategy has to be chosen accordingly.

4.1 Image-based Rendering

To overcome the difference in frame rate, which results from decoupling the rendering process, we utilize image-based rendering techniques. Per pixel depth information

can easily be stored during the rendering process. In combination with the matrices used for rendering and the current camera matrix, we can re-project every pixel to derive a novel view of the scene. Some issues related to this approach have to be addressed in order to produce satisfying results. The first issue is one we have already encountered with super-resolution of the video stream. Since image warping uses a static image from a different viewpoint than the current, disocclusion is again a topic. A common way to address this issue is to cache or even render different views in a preprocessing step [8]. In addition to the latest rendered image, an appropriate stored view can be used to fill disoccluded areas. In contrast, we employ hole filing strategies to account for this issue.

Not only disocclusion, but also occlusion is a problem, which needs to be handled. If parts of the rendered object occlude other parts due to the change in camera position, different pixels are warped to the same location. This results in a *depth fighting* like behavior. Another problem that comes with this approach is sampling related. If the camera in the derived view is closer to the scene than in the original one, the image is sampled at a higher rate, leading to cracks in the novel view.

The straightforward way for implementing image warping would be to multiply each pixel and its depth position by the inverse projection and model matrix used for rendering and then again by the projection and model matrix of the current view, like given in equation 5, where P is the projection matrix, C_{ref} is the camera matrix used for rendering, $C_{current}$ is the camera matrix of the desired view and p is the position of the pixels given in homogeneous coordinates.

$$p' = P \cdot C_{current} \cdot C_{ref}^{-1} \cdot P^{-1} \cdot p \tag{5}$$

The problem of this approach is that it is likely that different pixel are projected onto the same output position. Since we use parallel warping on the GPU, this results in undefined behavior. We implemented the image warping in the following way to resolve the described problems: In a first step, only the depth value of every pixel is warped into the desired view. We resolve the described race condition by using atomic operations to store only the depth values closest to the camera. This is similar to the z-buffer algorithm. In the next step, the depth and position of each pixel is used to look up the color in the original image. This is done by performing the operation given in equation 5, in the reverse direction. Using this multi-stage warping, we can efficiently eliminate resampling artefacts and occlusion.

4.2 Hole Filling

In the final step, we aim to fill small holes and cracks which result from the image warping. The simple approach is to interpolate holes from neighboring pixels iteratively or using push-pull interpolation [12].

Another approach is to re-render the disoccluded areas to fill the holes an cracks. To detect disoccluded regions, we first render only the silhouette of the model from the current viewpoint. Second, we warp the image and subtract the result from the silhouette which gives us the disoccluded region. Now, the pathtracer is instructed to recompute only the identified area with a small number of samples in order to keep the computational expense low. Thus, we can efficiently fill all holes with the disadvantage of discontinuities between the original rendering and the re-rendered area due to the small sample count. Figure 5 shows a comparison of the two hole filling methods. The re-rendering produces good results in any case, while the computational expense is again related to the size of the disoccluded area and is higher compared to interpolation. The interpolation works well as long as the holes are small.

5 Results

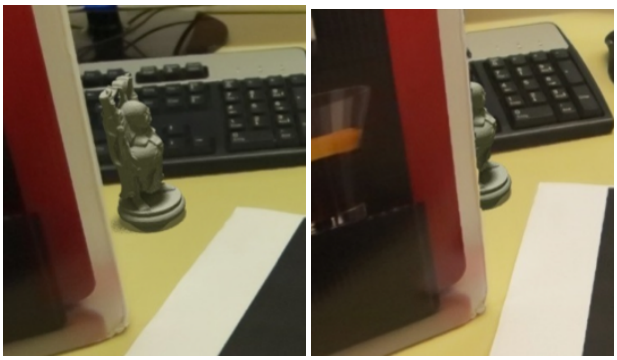


Figure 6: Occlusion of a virtual object (Buddha statue) by the real world using the estimated depth information.

We used three different scenarios to evaluate the SR method. Two of the data sets show office scenes, while one shows an outdoor scene. All scenes have different complexity in terms of depth, depth range and texture. As metric for similarity, Hdr-vdp-2 is used [11] to show that the results of our algorithm are similar to the reference image and free of artifacts. To account for image sharpness, the reference free LPC-SI [7] metric is used. The input video stream was captured at 640x480 pixels (0,3 megapixels) with 30 frames per second (fps) while the HR still images had a resolution of 2304x1728 (4 megapixels) pixels captured with 8 fps. The system was evaluated on an Intel Core i5 (3.4 GHz) with 8 GBs of memory and a Geforce GTX 680. For capturing the test data, a Canon IXUS 240 HS was used. Figure 7 shows average scores for Hdr-vdp-2 (similarity to the ground truth in percent) and LPC-SI (higher score means more sharpness) of all three scenes. It can be seen that the sharpness of our method is close to the ground truth and way above what bilinear

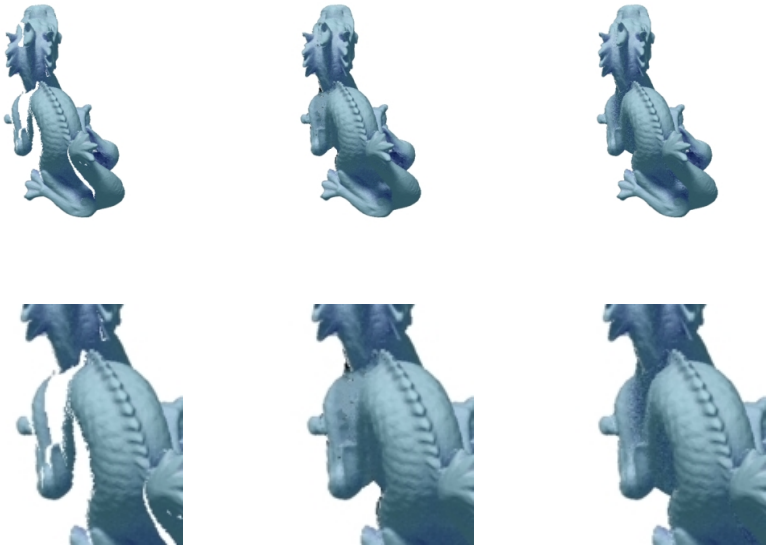


Figure 5: Comparison of the two hole filling methods for different warping angles. The first column shows the warped image without hole filling, the second with color interpolation and the third with re-rendering of disoccluded areas. The bottom row gives a detail view.

interpolation gives us. The online SR resolution runs at 7 fps (average) on the reference system.

Using the estimated depth, we can now let real objects in the scene occlude augmented virtual objects to generate a more immersive AR experience. This is done using an CUDA kernel, which compares the depth of the virtual scene with the estimated depth of the real scene and blends the images accordingly. Figure 6 shows an exemplary result. The depth estimation runs at 12 fps on average. Since the optical flow gives good sub-pixel accurate correspondences, the resulting depth map features sharp depth discontinuities, which is crucial for artifact-free occlusion.

Limitations The super-resolution part of our work is applicable on static and dynamic scenes. It has to be kept in mind that fast motion of both camera and objects in the scene leads to bad image registration results. Due to our flow quality measure, the algorithm does not fail but rather falls back to the LR video stream. In case of fast motion the fall back might only be hardly noticeable due to motion blur. In the case of slow motion results show that high frequency details can be preserved nicely by our algorithm. While image-based rendering in general is a very versatile technique, a limit is encountered, when it comes to view dependent effects. Effects like specular lighting, refraction and parallax depend on the position of the viewer relative to the scene.With our technique those effects cannot be simulated, since all information is baked into the rendering. Therefore, the quality of the result drops dramatically, if view dependent effects are simulated.

6 Conclusion

We developed a system capable of improving the overall quality of an augmented reality setup. Quality in this case means on the one hand the spatial resolution of the video stream and on the other hand the visual quality of the augmented information. Still images captured at a slow frequency are used to improve the spatial resolution of the low resolution video stream. By registering the high resolution still image with sub-pixel accuracy, we can warp it to fit the current video frame. Furthermore, we designed a reference-free metric for the quality of the optical flow that lets us decide whether the image registration was successful or not. Based on this metric, we blend the high resolution image with the video stream. As a result, we get a method that can upsample videos from 640x480px to 2304x1728ms in under 200ms. The algorithm in the worst case falls back to the quality of the LR video stream, introducing only small artifacts in extreme cases. The evaluation shows that this approach yields good results in a variety of scenarios. Using the optical flow to solve the correspondence problem, we can compute a depth map of the scene, which can further be used to create immersive AR effects.

The computational complexity of physically correct rendering algorithms like pathtracing is high, which does not allow for interactive rendering, especially if the spatial resolution is also high. We described approaches to decouple the frequency of the rendering process from the displaying frequency by distributing the workload over multiple frames. The resulting difference in frame rate is then addressed using image warping. The results show that this strategy can be applied to move complex rendering algo-

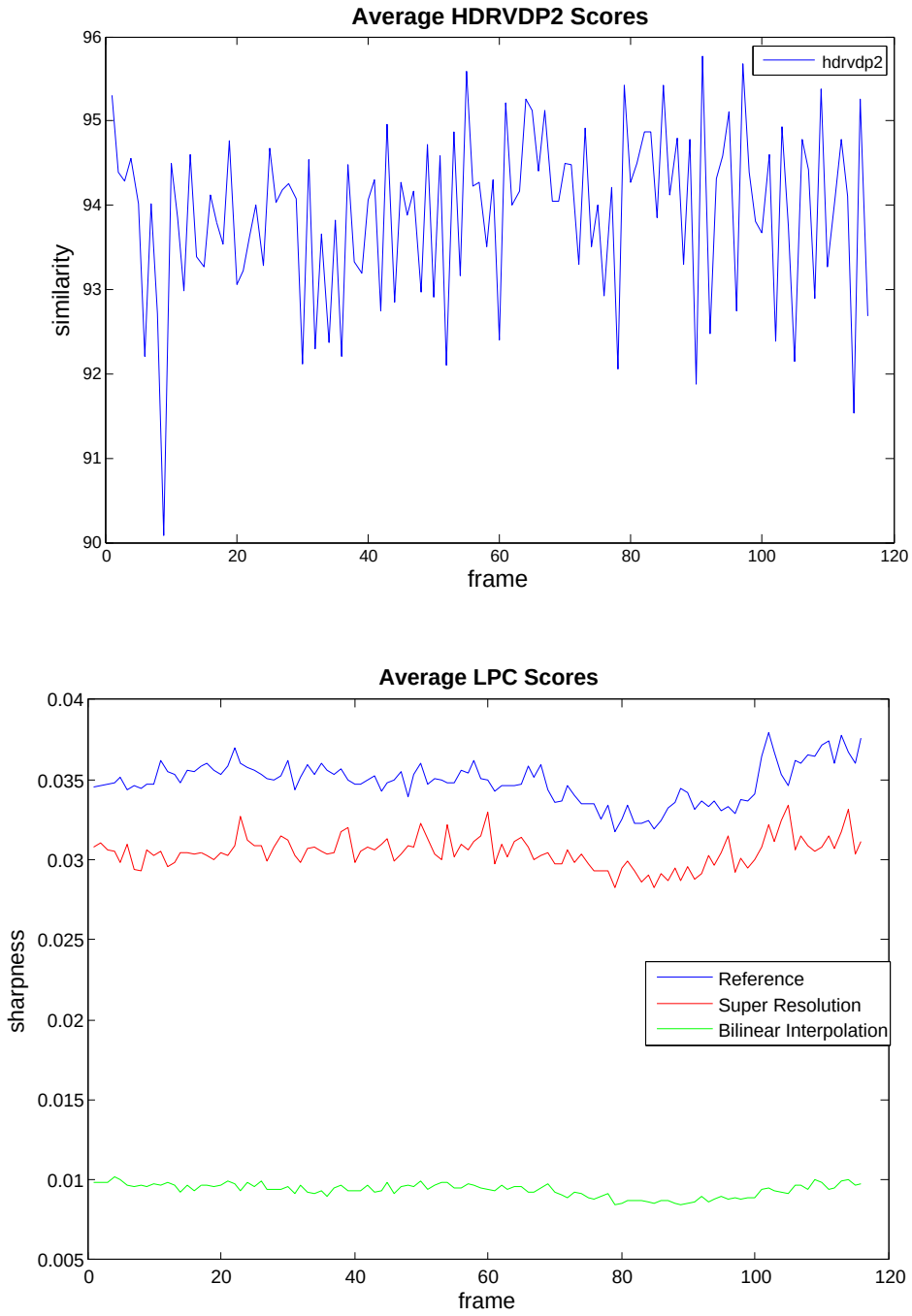


Figure 7: Average results for Hdr-vdp-2 (similarity) and LPC-SI (sharpness). The upper figure shows that our result is close to the ground truth in terms of structural similarity. The lower figure displays that the output of our SR method produces a sharp high resolution version of the input data close to the ground truth and superior to a bilinear interpolated version.

rithms towards interactivity without a noticeable loss of quality.

Future Work The sub-pixel accurate image registration step is achieved by computing the optical flow between the still frame and the current video frame. This has to be done twice each frame, since we need the forward and the backward flow to be able to compute our quality metric. This step is the most time consuming part of the algorithm. Since we know the camera pose from tracking, the flow computation can be simplified to a one dimensional problem by using epipolar geometry. This simplification would not only speed up the computation, but also might give better results in terms of quality, since the search space is significantly smaller. Furthermore, we aim to move the super-resolution algorithm of our work to mobile devices. Especially for this case, the image registration has to be simplified to achieve a satisfying performance.

References

[1] Simon Baker and Takeo Kanade. Hallucinating faces. In *Automatic Face and Gesture Recognition, 2000. Proceedings. Fourth IEEE International Conference on*, pages 83–88. IEEE, 2000.

[2] John L Barron, David J Fleet, and Steven S Beauchemin. Performance of optical flow techniques. *International journal of computer vision*, 12(1):43–77, 1994.

[3] Pravin Bhat, C Lawrence Zitnick, Noah Snavely, Aseem Agarwala, Maneesh Agrawala, Michael Cohen, Brian Curless, and Sing Bing Kang. Using photographs to enhance videos of a static scene. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 327–338. Eurographics Association, 2007.

[4] Piotr Didyk, Elmar Eisemann, Tobias Ritschel, Karol Myszkowski, and Hans-Peter Seidel. Perceptually-motivated real-time temporal upsampling of 3d content for high-refresh-rate displays. In *Computer Graphics Forum*, volume 29, pages 713–722. Wiley Online Library, 2010.

[5] William T Freeman, Thouis R Jones, and Egon C Pasztor. Example-based super-resolution. *Computer Graphics and Applications, IEEE*, 22(2):56–65, 2002.

[6] William T Freeman, Egon C Pasztor, and Owen T Carmichael. Learning low-level vision. *International journal of computer vision*, 40(1):25–47, 2000.

[7] Rania Hassen, Zhou Wang, and Magdy Salama. No-reference image sharpness assessment based on local phase coherence measurement. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 2434–2437. IEEE, 2010.

[8] Stefan Hauswiesner, Denis Kalkofen, and Dieter Schmalstieg. Multi-frame rate volume rendering. In *Proceedings of the 10th Eurographics conference on Parallel Graphics and Visualization*, pages 19–26. Eurographics Association, 2010.

[9] Robert Herzog, Elmar Eisemann, Karol Myszkowski, and H-P Seidel. Spatio-temporal upsampling on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 91–98. ACM, 2010.

[10] T. S. Huang and R. Y. Tsay. Multiple frame image restoration and registration. In *Advances in Computer Vision and Image Processing*, volume 1, pages 317–339, Greenwich, 1984.

[11] Rafal Mantiuk, Kil Joong Kim, Allan G. Rempel, and Wolfgang Heidrich. Hdr-vdp-2: A calibrated visual metric for visibility and quality predictions in all luminance conditions. *ACM Trans. Graph.*, 30(4):40:1–40:14, July 2011.

[12] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. Efficient point-based rendering using image reconstruction. In *SPBG*, pages 101–108, 2007.

[13] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 39–46. ACM, 1995.

[14] Leonard McMillan Jr. *An image-based approach to three-dimensional computer graphics*. PhD thesis, Citeseer, 1997.

[15] M. Otte and H.-H. Nagel. Optical flow estimation: Advances and comparisons. In Jan-Olof Eklundh, editor, *Computer Vision ECCV '94*, volume 800 of *Lecture Notes in Computer Science*, pages 49–60. Springer Berlin Heidelberg, 1994.

[16] Pitchaya Sitthi-amorn, Jason Lawrence, Lei Yang, Pedro V Sander, and Diego Nehab. An improved shading cache for modern gpus. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 95–101. Eurographics Association, 2008.

[17] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. In *Rendering techniques 99*, pages 19–30. Springer, 1999.

[18] Xuan Yu, Rui Wang, and Jingyi Yu. Real-time depth of field rendering via dynamic light field generation and filtering. In *Computer Graphics Forum*, volume 29, pages 2099–2107. Wiley Online Library, 2010.

Parallel Graphics

Impact of Modern OpenGL on FPS

Jan Čejka*
Supervised by: Jiří Sochor†

Faculty of Informatics
Masaryk University
Brno/ Czech Republic

Abstract

In our work we choose several old and modern features of OpenGL that applications use to render scenes and compare their impact on the rendering speed. We aim our comparison not solely on these features, but also on the type of hardware used for the measurements. We run our tests on a professional graphics card QUADRO 6000 and on a consumer graphics card GeForce GTX 580, and evaluate how actual hardware influences the results.

Keywords: OpenGL, Core profile, QUADRO, GeForce

1 Introduction

Graphics hardware is in constant evolution. New models come with new methods to solve the same problems in a more effective way. Developers now stand before a dilemma of whether to use an old method that had more time to be optimized in drivers, or a new method that better uses new technology but is not well established and may actually slow down the application when used inappropriately.

When analysing a well known graphics library OpenGL, we found that in its more than twenty years of evolution it really accumulated multiple solutions for the same problems. Considering for example rendering commands, we may use a pair of *glBegin* and *glEnd* commands and define geometry vertex by vertex, use *glDrawElements* and draw multiple primitives in a few commands, use functions such as *glMultiDrawElements* to reduce the number of rendering commands even further, or use indirect draw commands to manage rendering entirely from the GPU itself. Moreover, in case of static geometries, we still have an option to pack these commands into display lists and again reduce the number of commands that need to be processed.

To design fast applications, developers must decide which of these methods to implement. Their decision is based not only on the type of application, but also on the properties and the architecture of the hardware they use, and of course, their own experience.

We also had to make this decision in our application VRUT. This application is developed in a cooperation between a number of universities in the Czech Republic and the automobile company ŠKODA Auto a.s. The name stands for Virtual Reality Universal Toolkit and its purpose is to visualize detailed geometry in real-time. As such, it is highly demanding on efficiency of rendering. It is used by students as well as professionals, and therefore, it runs on various kinds of hardware, which also influences the speed of rendering.

We extended VRUT with a new rendering module and implemented several techniques that solve fundamental problems in rendering. In this paper, we describe them and compare their impact on the resulting frame rate. As this frame rate is affected by hardware, we present results of testing on two different NVIDIA graphics cards, GeForce GTX 580 and QUADRO 6000.

This paper is structured as follows. The next section presents several works that analyse modern OpenGL and its features. The third section describes methods we chose and tested in our application. Results of these tests are presented and discussed in the fourth section. The final, fifth section concludes our work and emphasizes the most important points.

2 Related work

OpenGL specification [7], located at OpenGL website *www.opengl.org*, contains detailed description of all OpenGL 4.4 functions. This website also lists all available OpenGL extensions and their description in form of plain texts. In addition to this, some extensions are also described on sites of other companies that define their own extensions; for example, NVIDIA presents at *https://developer.nvidia.com/nvidia-opengl-specs* a list of extensions that are available on many NVIDIA graphics cards.

Features of new versions of OpenGL are often presented at the SIGGRAPH conference; Lichtenbelt [3] gives us additional information about version 4.4. However, some researches focus just on a part of OpenGL. McDonald and Everitt [4] describe how techniques introduced in OpenGL 4.3 can reduce the number of functions that need to

*xcejka2@fi.muni.cz
†sochor@fi.muni.cz

be called to render the whole scene. Gateau [1] presents techniques developed by NVIDIA to render complex objects in only a few (possibly one) draw calls. These works unfortunately lack thorough testing and do not take different hardware into account at all.

Much information about hardware can be found on the hardware manufactures' web pages intended for developers[1,2]. Additional information is given at conferences; Kilgard [2] describes features of NVIDIA's graphics cards in association with newest versions of OpenGL. A list of main features in which QUADRO professional graphics cards and GeForce gamer cards differ can be found in [5].

3 Analysed techniques

In our work, we chose and compared several techniques that solve problems of rendering scenes. We focused on drawing commands, the type of pipeline, vertex array setup and the rendering context used.

3.1 Draw commands

The current version of OpenGL supports two main methods of drawing primitives. The first method uses functions *glBegin* and *glEnd* and defines vertices separately. The other method stores all data of these vertices in arrays, and uses functions like *glDrawArrays* and *glDrawElements* to draw them all at once. OpenGL also improves the latter method and offers functions like *glMultiDrawElements*, allowing the packing of many *glDrawElements* calls into one.

We may also use display lists in addition to these methods. These display lists allow the driver to store all commands in the most effective way and then recall them when appropriate. This method is used mainly when rendering with *glBegin* and *glEnd* as it saves many function calls, but it can be used to draw with vertex arrays as well.

3.2 Fixed-function and programmable pipeline

OpenGL defines a set of operations that are applied to each processed primitive. These operations include transformation, lighting, texturing and many more, and form a pipeline. First versions of OpenGL defined a set of fundamental operations; to use them, programmers needed to set their parameters and activate or deactivate them if necessary. This is called fixed or fixed-function pipeline.

With time, the number of operations in pipeline increased, and so did the number of their combinations. As such, setting these parameters became impractical. Since version 2.0, OpenGL allows some parts of its pipeline to be programmed by small programs called shaders. These shaders define which operations are performed on vertices

[1] developer.nvidia.com
[2] developer.amd.com

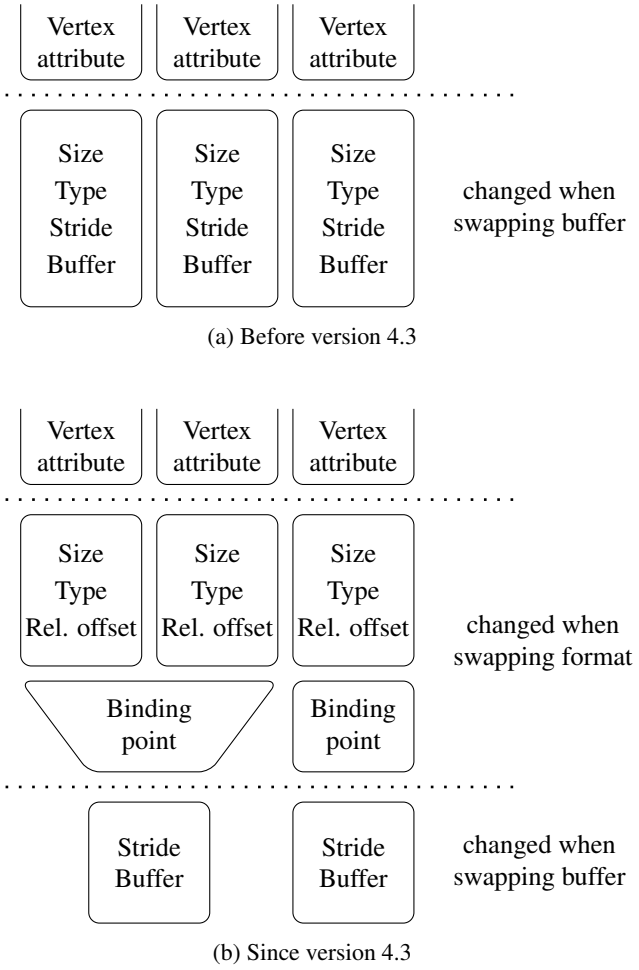


Figure 1: Setting vertex array parameters before OpenGL version 4.3 and since version 4.3

and fragments[3] and manage their order. This is usually referred to as a programmable pipeline.

When we render simple scenes with simple geometries, simple lights and simple materials, we do not need functionality of the programmable pipeline, as the fixed pipeline fulfills our needs. For this reason, we do not need to write and optimize complex programs of the programmable pipeline and thus we save some time and effort. On the other hand, well written programs may save some costly state changes done in the fixed pipeline.

It is questionable which of these pipelines leads to a better performance in given situations. Most of modern graphics cards are programmable, and OpenGL's fixed functionality is programmed in the driver after all. Many modern programs (especially computer games) utilize shaders, which may lead driver programmers to dedicate less effort to optimizing drivers for programmable pipeline in comparison to fixed pipeline. On the other hand, fixed pipeline exists since the first version of OpenGL and had

[3] There are also geometry shaders which operate on primitives and tessellation shaders which subdivide them, however, our application uses only vertex and fragment shaders.

Figure 2: Scene with the Fabia car used when measuring rendering speed

more years to get optimized for applications that uses it.

3.3 Buffer setup

OpenGL 1.1 came with vertex arrays and allowed to process multiple primitives with a single call. It provided a few functions to set parameters of these arrays, that is the size and the type of vertex attributes, a buffer with these data and the stride between them, but all of them had to be set when an attribute or a buffer changed, as illustrated in Figure 1a.

For many years, this was the only method to set up buffers. In 2012, OpenGL 4.3 reviewed when vertex array parameters are set and designed a technique that decreased the amount of data set each time. It introduced binding points, which are places where buffers can be bound, and allowed us to bind buffers and set attribute parameters separately, as illustrated in Figure 1b.

In addition to this, vertex array objects (VAOs), presented in OpenGL 3.0, allow to create objects holding all information about the setup of vertex arrays. They are similar to buffer objects or texture objects, which have both existed in OpenGL for many years. VAOs do not change the way vertex arrays are set up. They only make switching between them easier.

We tested and compared both methods of setting vertex arrays and VAOs. Moreover, we also decided to test an extension *NV_vertex_buffer_unified_memory* developed by NVIDIA, available on their graphics cards. This extension is described in [6], and its main idea lies in querying the address of memory allocated by vertex buffer objects. Using this address in plane of a vertex buffer allows us to save the driver some work which speeds up the rendering.

3.4 Rendering context

The last issue we focused on is aimed at the OpenGL context. There are many parameters that are set when creating this context. We tested two of them, the type of profile and the presence of debug features, and measured their influence on the speed of rendering.

OpenGL profiles were introduced in version 3.2 as a form of removing deprecated functions. OpenGL defined two of them: core and compatibility. The core profile contains only the most modern features, while the compatibility profile includes all functions since the first version of OpenGL. As the core profile could be simpler to implement and optimise by drivers, we decided to test, whether it leads to an increase in the number of rendered frames per second.

Like many other libraries, OpenGL comes with new methods to ease debugging and development of new applications. In addition to querying OpenGL for simple errors, some implementations allow us to create a debug context, giving us an option to set up callbacks that are called every time an error occurs. Using this debug context must obviously lead to a decrease in speed of rendering, as it must handle not only these errors, but also all attached callbacks. For this reason, we decided to measure its impact on the actual speed of rendering.

4 Measurement

We measured the time our application needed to render a single frame depending on several settings.

The measured scene contained a static model of the Fabia car containing 4.6 million triangles, illustrated in Figure 2. This model was represented by a hierarchy tree

Configuration		GeForce		QUADRO	
		CPU [ms]	**GPU** [ms]	**CPU** [ms]	**GPU** [ms]
1)	**B/E**	262.662	274.890	208.861	337.591
2)	**VA**	199.759	261.961	15.776	77.987
3)	**VBO**	25.876	243.702	6.624	26.397
4)	**DL + B/E**	327.224	333.550	4.154	8.599
5)	**DL + VA**	341.850	343.046	7.368	11.481
6)	**DL + VBO**	340.475	342.296	7.403	11.480
7)	**VBO + Shaders**	27.233	26.416	27.932	27.591
8)	**DL + VBO + Shaders**	487.704	485.719	10.083	22.440
9)	7) + **VAO**	27.711	27.281	28.644	27.816
10)	7) + **MDE**	11.568	16.916	11.238	21.172
11)	7) + **Bindless**	13.499	16.717	13.973	21.260
12)	7) + **MDE + Bindless**	9.616	16.964	9.828	21.148
13)	7) + **VAO + MDE**	11.170	16.649	10.488	21.183
14)	7) + **VAO + Bindless**	14.659	16.764	14.776	21.301
15)	7) + **VAO + MDE + Bindless**	10.630	16.754	10.584	21.206
16)	7) + **MDE + Format43**	11.456	16.742	11.210	21.186
17)	7) + **VAO + MDE + Format43**	10.746	16.723	10.798	21.212
18)	**Core**	27.953	27.445	28.369	28.001
19)	12) + **Debug**	33.244	31.384	—	—
	4) + **Debug**	—	—	5.043	8.775

Table 1: Rendering times of different configurations on both tested machines

with 1344 geometry nodes, containing 1342 triangle lists and 145830 triangle strips. The model's appearance was described by 86 materials; one of them implemented a car paint effect and used its own shaders, the rest were simple enough to be rendered with the fixed pipeline. The scene was lit by a simple directional light centered at the camera (headlight).

We ran all tests on two machines. The first machine, in the rest of the paper labelled as **GeForce**, contained Intel i7 2600 processor with 8 GB of main memory and GeForce GTX 580 with the display driver version 310.70. It was chosen to represent consumer machines.

The other machine, labelled as **QUADRO**, contained two Intel Xeon X5680 processors and 24 GB of main memory. It had two QUADRO 6000 graphics cards with the display driver version 310.70. It was chosen to represent professional workstations. Although this machine contained two graphics cards, only one of them was active so that the results could be compared with the results of the first machine.

We measured a time these machines needed to render the scene in configurations described below. Since the actual rendering runs asynchronically on graphics cards, we separately measured the time needed to call all OpenGL functions (labelled as **CPU**) and the time the driver needed to execute and complete all commands (labelled as **GPU**). We measured groups of 50 frames and chose three groups with the smallest deviation. Since we focused on the maximum speed of rendering, we took the minima of measured values (in milliseconds) and presented them in Table 1.

4.1 Configurations and Results

We tested the following configurations. First, we tested the influence of draw commands used to render the geometry. We compared rendering with *glBegin* and *glEnd* functions (in the table labelled as **B/E**), rendering with vertex arrays stored at the client site (**VA**), and rendering with vertex arrays stored at the server site (**VBO**). Since all these draw commands can be stored in display lists, we made the same measurement again, this time using display lists (**DL**). The results are shown in Table 1 with configurations numbered 1 – 6.

Next, we tested how shaders influence the speed of rendering (configuration labelled as **Shaders**). The largest part of the scene could be rendered using the fixed pipeline, therefore, we compared the rendering speed when using the fixed pipeline and the programmable pipeline with shaders. These shaders simulated operations of the fixed pipeline, however, we must note that they implement per-pixel lighting. Despite the fact they are more computationally demanding than the fixed pipeline, we believe the measured values are still comparable. The results are numbered as 7 and 8.

We also measured the influence of vertex array objects (labelled as **VAO**), *glMultiDrawElements* (**MDE**), *NV_vertex_buffer_unified_memory* extension (**Bindless**), OpenGL 4.3 technique of setting vertex arrays (**Format43**) and their combinations. We chose the configuration 7), that is **VBO + Shaders**, as a starting configuration for this group of configurations. In the Table, these configurations are numbered as 9 – 17.

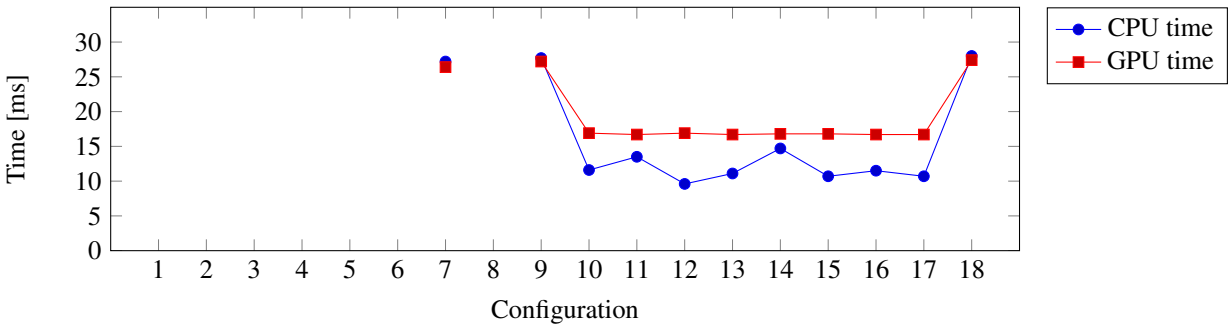


Figure 3: Rendering times of configurations achieved on **GeForce**. Times of configurations 1 – 6 and 8 are too large to display

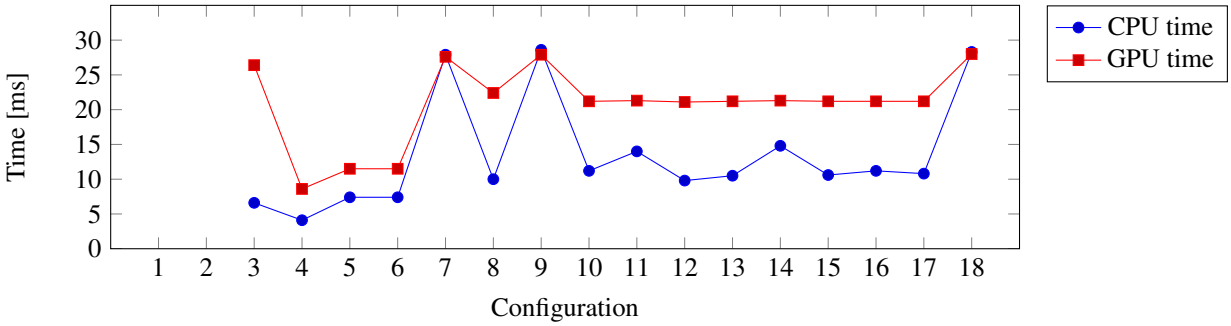


Figure 4: Rendering times of configurations achieved on **QUADRO**. Times of configurations 1 and 2 are too large to display

Finally, we tested the influence of the rendering context parameters. Rendering with the core profile (labelled as **Core**) is numbered as 18, and automatically implies **VBO + Shaders + VAO** are active. For the debug context test (labelled as **Debug** and numbered as 19), we chose as the starting configurations the best configuration on each machines, that is the configuration 12 on **GeForce** and 4 on **QUADRO**.

Figures 3 and 4 shows rendering times as line graphs for the **GeForce** machine and the **QUADRO** machine respectively. Rendering times greater than 50 ms are not shown, so that the difference in time of other configurations could be better visible. Also, configuration 19 is not present, because it differs between both machines, and the figure could lead to a misinterpretation of results.

4.2 Discussion

The measurement revealed several interesting facts. Measured rendering times in configurations 1 – 8 show, that using shaders in cooperation with vertex arrays stored at the server side in vertex buffer objects is essential for fast rendering on the **GeForce** machine. On the other hand, activating display lists led to a severe performance hit. We believe this happened due to the fact that GeForce graphics cards (as well as other consumer graphics cards) are optimized for computer games that usually do not contain static geometries and use modern features of graphics libraries, especially the programmable pipeline.

However, this was not true for the **QUADRO** machine, where display lists were the most effective way of rendering geometry. This is probably the result of driver optimizations, since display lists are a perfect solution for static geometries present in many professional applications.

Configurations 9 – 17 give us more information about the contribution of other modern techniques to the speed of rendering. Using vertex array objects showed a slight slow down when compared to configuration 7. This could have been caused by misunderstanding the role of these objects leading to an improper implementation in our rendering module, or by insufficient optimization in the driver.

Using extension *NV_vertex_buffer_unified_memory* and *glMultiDrawElements* led to an increase in speed of rendering. It is obvious that setting buffers and calling draw commands were probably the bottlenecks in our application, and these features effectively reduced their impact on the final speed of rendering. OpenGL 4.3 technique to set up vertex arrays did not result in any significant speedup. We should also mention an interesting fact, that all configurations 10 – 17 show approximately the same GPU time, but they differ in the time the application needed to call all functions.

The last two configurations show that using the core profile does not lead to any significant difference in performance (compare configurations 9 and 18). Obviously,

debugging with debug context causes a performance loss. This loss was much smaller on the **QUADRO** machine. We, therefore, assume that professional graphics cards are more optimized and therefore more suitable for debugging than consumer graphics cards.

5 Conclusion

We measured and compared the speed of rendering of a static scene in several configurations depending on techniques used and the type of hardware. We found that using modern features of OpenGL such as shaders and vertex buffer objects led to an increase in the rendering speed on NVIDIA's consumer graphics card GeForce. On the other hand, professional graphics card NVIDIA QUADRO achieved the best rendering times when we used display lists and the fixed function pipeline. Given these results, we believe neither old nor modern features are the absolute choice for better rendering performance, as this highly depends on the hardware the application runs on.

Acknowledgments

We would like to thank ŠKODA Auto a.s. for the model of the Fabia car, and also Antonín Míšek for his ideas and measuring on the **QUADRO** machine.

References

[1] Samuel Gateau. Batching for the masses: One glCall to draw them all. SIGGRAPH, 2013.

[2] Mark J. Kilgard. NVIDIA's OpenGL functionality. GPU Technology Conference, 2010.

[3] Barthold Lichtenbelt. Announcing OpenGL 4.4. SIGGRAPH, 2013.

[4] John McDonald and Cass Everitt. Beyond porting. Steam Dev Days, 2014.

[5] NVIDIA. *NVIDIA Quadro vs. GeForce GPUs: Features and Benefits*, 2003.

[6] NVIDIA. *OpenGL Bindless Extensions*, 2009.

[7] Mark Segal and Kurt Akeley. *The OpenGL® Graphics System: A Specification*, 2013.

Parallelization of Shape Diameter Function Computation using OpenCL

Rastislav Kamenicky*
Supervised by: Martin Madaras†

Faculty of Mathematics Physics and Informatics
Comenius University
Bratislava / Slovakia

Abstract

Shape Diameter Function (SDF) is a scalar function that expresses a measure of the diameter of the object's volume in the neighborhood of each point on the surface on an input mesh. It is fundamental in many applications in computer graphics used for consistent mesh partitioning and skeletonization. The algorithm sends several rays inside a cone centered around the point's inward-normal direction and measures the distance at the point of intersection. We have implemented the original algorithm and further extended it on GPU by parallelizing the ray casting process using OpenCL. We have also generalized the algorithm to support non-manifold meshes. The algorithm shows great speedup in terms of timing when compared with the CPU based implementation.

Keywords: Shape Diameter Function, OpenCL, Parallelization

1 Introduction

Analysis of 3D models and processing of spatial data is a fundamental part of computer graphics. However acquired models are often non manifold or lack crucial data i.e. skeletal representation, UV coordinates, etc. Methods as mesh processing and shape analysis are commonly used to fill missing information. Such methods require algorithms that are robust and work fast and effectively.

SDF is a volume-based shape function that can help to process and manipulate families of objects which contain similarities using a simple and consistent algorithm. It can be used for skeleton extraction and mesh partitioning and contraction. SDF remains largely unaffected by pose changes of the same object and maintains similar values in analogue parts of different objects [16]. The diameter measured also relates to the medial axis transform (MAT) [4]. However, unlike the expensive computation and handling of medial axis, SDF is much simpler. It is a scalar field created by sending several rays from every input point on the mesh, measuring the distance at the point of intersection.

Such ray casting is highly parallel algorithm. If processed on the CPU, the task becomes extremely inefficient. Therefore, in our approach instead of tracing one ray at a time, we propose a parallel method for computing SDF that is performed on GPU using OpenCL, exploiting the independence of rays.

The method was originally meant to be used effectively only on manifold structures, but the process could be natively expanded to non-manifolds. And so we propose several changes that could further improve it's support for non-manifold structures. Finally, at the end of this paper, we compare the results of our GPU implementation against the CPU implementation.

Figure 1: Visualization of Shape Diameter Function with values normalized to interval $\langle 0, 1 \rangle$.

*kamenicky8@uniba.sk
†madaras@sccg.sk

2 Related Work

MeshLab implementation, implemented by Baldacci [2] used a different approach for calculating SDF. The method is based on iteratively peeling two or three successive depth layers of the mesh from multiple views around the mesh. This is performed through shaders and uses native GPU support for mesh projection and rasterization. Thanks to uniform memory access, this could achieve better results on larger meshes, but it loses detail on parts that are too close to the camera.

In [6], it is pointed out that SDF can be approximated using only a small subset of data. The remaining data is interpolated via Poisson interpolation. Even though the speedup is very significant, a lot of detail is lost on complex surfaces and areas where different body parts connect. This information can be crucial when connecting parts of skeletons and could lead to improper skeletonization.

In [15], it is mentioned that the outliers removal technique proposed in [16] generates counterintuitive results in some cases. The SDF value calculated by the Shapira et al. method is given by the weighted average of all the values thrown inside the point's cone, which for example in the case of a mesh composed of two parallel (infinite) planes, underestimates the correct diameter due to large cone size. In [15] to resolve the dilemma between a small or large cone, a more conservative estimation of the SDF is introduced by using an adaptive cone size. In the case, for example, of the infinite parallel planes this method converges to a very small cone size giving a correct SDF value equivalent to the distance between the two planes. In our GPU implementation we maintain the original outliers removal approach proposed in [16], leaving the one proposed by [15] for future work, because the adaptive cone size is computationally more expensive than original method. The ray has to be cast multiple times to find the correct cone size.

3 Original SDF Algorithm

Let M be an input mesh surface defining a volumetric object. SDF is a scalar function $f_v: \mathbf{M} \rightarrow \mathbb{R}$ that consists of creating a cone centered around inward-normal direction (the opposite direction of its normal) of every point $\mathbf{p} \in \mathbf{M}$. Inside this cone several rays are sent to the other side of the mesh, measuring the euclidean distance at the point of intersection. Outliers are removed and the remaining values are averaged and smoothed. As a result there is a single value for every point $\mathbf{p} \in \mathbf{M}$. The original algorithm consists of 4 steps.

Step 1 - preprocessing: In order to facilitate the ray casting, an acceleration structure is needed. Therefore in preprocessing stage an octree is created.

Step 2 - ray casting: In the ray casting stage rays are cast through octree and euclidean distance at the point of intersection is measured. According to Shapira et al. [16], the ideal number of rays is 30 inside a cone with angle of 120°. The rays are chosen randomly.

Step 3 - outliers removal: After the measured distances are obtained, the rays that are in the same direction as the inverse normal of the mesh they hit (the same direction is defined as an angle difference less than 90°) are ignored. This is performed to remove false intersections with the outside of the mesh. The SDF at a point is defined as the weighted average of all rays lengths which fall within one standard deviation from the median of all lengths. The weights used are the inverse of the angle between the ray to the center of the cone. This is because rays with larger angles are more frequent, and therefore have smaller weights.

Step 4 - smoothing: In order to increase robustness and fill in the values for points that could have ended up with 0 valid rays a smoothing stage is necessary. Anisotropic smoothing is chosen to smooth the values of the points on the mesh.

4 Our Implementation

We have based our implementation upon the original algorithm extending it on GPU by parallelizing the ray casting process using OpenCL. We have inherited all the steps from original algorithm and further extended them on GPU.

4.1 Preprocessing

In the preprocessing stage we have to create an acceleration structure around the mesh that will improve the ray tracing routine. The acceleration structure is one of the most important parts of ray tracing. As noted by [5], the fastest acceleration structure for static scenes is kd-tree, followed by bounding volume hierarchies (BVH). However for the purpose of comparison with the original article, we have chosen to use an octree to be able to compare results, leaving the other ones for future work. We computed the octree on CPU because the preprocessing time was not a concern. It is built in a top-down manner. Triangles that were in the middle of several nodes were detected through Mollers AABB-triangle intersection algorithm [1] and split into multiple nodes. On our test models optimal octree depth ranged from 6 to 12 depending on the number of triangles. We have chosen a maximum depth of 10 for consistent results.

4.2 Ray Casting

There are several ways to perform ray casting on GPU. Carr et al. [3] proposed to generate rays and perform traversal of acceleration structures on CPU, then store the results and perform ray-triangle intersections on GPU. Purcell et al. [14] proposed to store scene geometry and acceleration structure on GPU and perform both traversal and intersection on GPU. Recent efforts in optimizing the algorithms for GPUs have demonstrated to obtain better results on traversal of acceleration structures than a single-core execution on CPU. Therefore in our approach we send data containing triangle indices and acceleration structure to GPU, then we perform a per-ray computation of SDF. Our code is based upon the original implementation of the algorithm extending it to GPU with the help of OpenCL [12]. OpenCL was chosen because it is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, making it an universal solution for multiple platforms.

The computation of SDF is a three step process that consists of generating and casting rays, traversing the chosen acceleration structure and measuring the distance at the point of intersection with the geometry.

Ray Generation: In the first step, we have to generate N rays in a cone centered around inward-normal direction of a given point p. The generation of rays on CPU and then transferring the data to GPU creates unnecessary overhead because we have to store every ray and it's associated weight (inverse of the angle between the ray and the center of the cone). Therefore the rays have to be generated on GPU, but original algorithm generates the rays randomly, which would require defining a pseudo-random generator in OpenCL and store weight per every ray. Rolland [15] has tackled this problem by defining a cone sampling strategy consisting of random rays that are uniformly generated inside the cone. However, we wanted a fast deterministic algorithm that would be uniform for both smaller and larger number of rays and for any given cone. This is to prevent storing weights and to avoid unnecessary bias in the values on the mesh, which can be seen in Figure 2. Therefore, we have decided to evenly distribute rays in a cone with the help of Spherical Fibonacci [10].

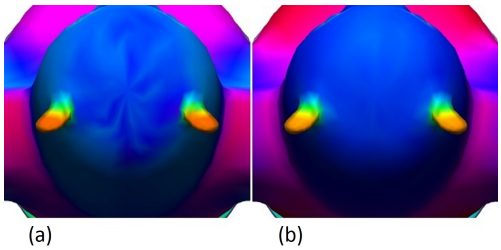


Figure 2: (a) random ray generation, (b) uniform ray generation using Spherical Fibonacci.

The algorithm generates the rays in a sphere from top to bottom. The generation was restricted to a given sphere cap and once a ray is generated, it is transformed into world coordinates by multiplication it with tangent-binormal-normal matrix specified by the point's inward-normal and two orthogonal unit vectors spanning the tangent plane of the point p. It is important to note that only valid points are used to generate the rays from (valid point is defined as having a non-zero normal, tangent and binormal vectors). Using triangle centers can have advantages over vertices in non-manifold models, where the normal of some points can not be properly determined. And it helps to reduce necessary data transfer thanks to the fact normal, tangent and binormal can be calculated.

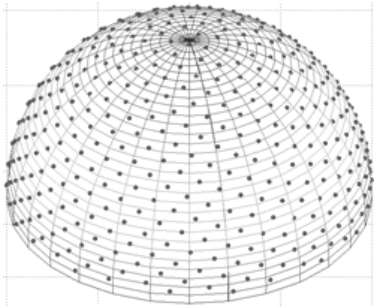


Figure 3: Generating rays uniformly with Spherical Fibonacci, image from [10].

Octree Traversal: In the second step we traverse our acceleration structure. Every ray is computed separately, one ray per work item. This is faster on modern cards that have thousands of cores. Unfortunately, we could not avoid random memory access that slows down the entire process. The octree structure and triangles are sent to GPU and each ray is cast separately. Several octree traversal methods are mentioned by Kristof et al. [7], notably neighbor pointer, kd-restart and short-stack approaches. Our implementation is based on Laine et al. [8] stack-based approach, but unlike [8] the tree is traversed in a top-down fashion all the way to the leaves. The nodes are traversed until we find a valid intersection in the third step.

Triangle Intersection: In the third step we compute ray-triangle intersection between the cast ray and triangles that belongs to a given node. As mentioned by Philippe et al. [13], one of the best algorithms for ray-triangle intersection is Moller and Trumbore [11] because it uses mainly dot and cross product that is fast on current graphical hardware. To improve the algorithm on non-manifolds, we have to skip intersections that are too close to the ray origin, like in the case of self intersecting mesh. We define minimum_closeness using the max dimension of model (max_size) as $mininum\_closeness = max\_size * 2.0 * 0.00001$; Once the intersection is found, we check if the ray is valid by comparing normal at the point of intersection and the ray direction as mentioned

in original algorithm [16]. In a case when no intersection is found, the distance is set to -1 and the ray is ignored. Afterwards the measured distance is stored in memory.

Data Management: Mesh triangles are stored in a 32bit RGBA texture. Coordinates (X, Y, Z, W) of each vertex of a given triangle are stored in a single texel. Normals and other necessary information is calculated on GPU. We encode the topology of the octree using 2 arrays. First one contains 32-bit child descriptors, each corresponding to a single node. The child descriptor contains a 24-bit child pointer and a 8-bit bit-mask that tells whether each of the child slots actually contains a node. In a case when the node is a leaf the child pointer points to the second array containing data for leaf nodes. Each leaf has to store the number of triangles it contains and their pointers to the triangle texture. This is all stored in the second array which can be seen in Figure 4.

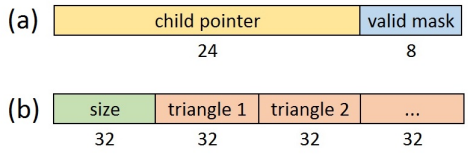


Figure 4: (a) 32bit child descriptor, (b) leaf data.

In a case when we are using triangle centers as the points from which we are casting the rays and measuring the distance, no additional information is necessary. Otherwise, we have to store the point's origin, normal and tangent or other information like triangle / vertex neighbors from which we can fill in the data. At last we have to store our results. They are stored in a single array with size = number_of_points $*$ number_of_rays. Figure 5 shows an example of how the octree topology is stored in memory.

4.3 Outliers Removal

Once we collected all values for our points and their rays, another program is executed on GPU. We do not have to send the data to GPU because they are already there from previous step. For output we create 1 array containing value for every point. The work is split in a way that 1 work item processes data of one point. Rays with lengths which do not fall within one standard deviation from the median of all lengths are removed and the rest is averaged using weights that can be calculated again thanks to our uniform sampling. After we get our final value, we send the data to CPU memory. While we are saving the data, we normalize them, this is very fast and does not needs to run on GPU because we would need another array for second output.

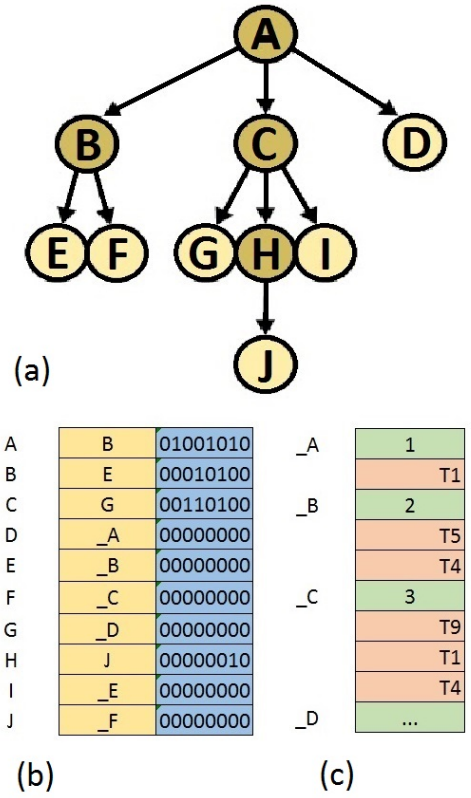


Figure 5: Octree topology on an example. (a) octree structure, (b) 1st array with octree nodes, (c) 2nd array with leaf data.

4.4 Smoothing

As mentioned in the original paper [16], to overcome errors in the measure caused by pose changes or complex surface geometry, a smoothing operation is necessary. The method chosen is directly related to the result we are trying to obtain because various methods can lead to significantly different values. Therefore we propose 3 approaches that can be used to smooth the SDF values in various ways.

Smoothing on Mesh: In first approach we smooth values in mesh, by defining k-ring neighborhood Gaussian smoothing. The k specifies the blur radius given by our connectivity in mesh, which can be seen in Figure 6. We start with a chosen vertex, for which the $k = 0$; In first iteration, we create a list of vertices that share an edge with our first vertex. These vertices have $k = 1$; In every next iteration we create a new list of vertices that share at least one edge with vertices from previous iteration, but only those that we have not yet chosen. This is repeated until we reach our desired k. We then create a 1D Gaussian matrix for our k and perform weighted averaging of all the values from vertices using data from our matrix as weights. This is repeated for every vertex in mesh. To ensure consistency and continuity during smoothing, duplicated vertices have to be merged into one. This ensures

that the k-ring neighborhood Gaussian smoothing will not fail to find neighboring vertices. This method of smoothing does not preserve values at corners. In a case when the values should be preserved, smoothing can be performed on triangles and the dihedral angle between the connected triangles from 2 consecutive iterations can be used as additional weight. This method can be parallelized, but it is not suited for GPU because it requires large dynamic arrays for the vertices and is mostly based on accessing memory than doing mathematical calculations.

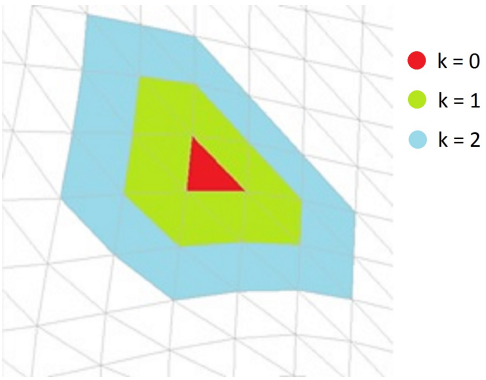


Figure 6: Smoothing by defining a k-ring neighborhood on the mesh.

Smoothing on Projected Points: Our next approach is based on projecting the points in their inward normal direction to a distance which is half of their SDF value. This creates a cloud like structure inside the mesh that resembles medial axis. Then for every point we perform nearest neighborhood search within the radius of the given point's SDF value, acquire the SDF value of every detected point and average the result. But due to the fact that many meshes have round, spheroidal parts where thousands of points can occupy small space, this would lead to a time complexity of $O(n^2)$ in worst case where n is the number of points. Therefore, for efficiency we have to join the points whose distance from each other is too small. This can be done by creating an octree structure, that will store the average value of projected points and their count in it's nodes. And instead of searching nearest points within the radius of the given point's SDF, we search nearest octree nodes, which can be seen in Figure 7. We perform this by traversing the tree from top to bottom, checking if nodes are within the radius of our SDF value. If a node is fully inside our radius, we do not traverse this node further. At the end, we average values from the nodes using weighted averaging. As weight we use the multiplication of number of points in each collected node and an approximate percentage of how much of the cube lies inside the radius. This method can be parallelized on GPU. Besides the standard octree structure that must be send to GPU, we have to include number of points and SDF value for every node. Leaves do not need to contain any additional pointers. Other information as the projected points and their SDF values can be obtained from data that remained from ray casting steps. Octree node positions and dimensions can be interpolated from the position and dimension of root node.

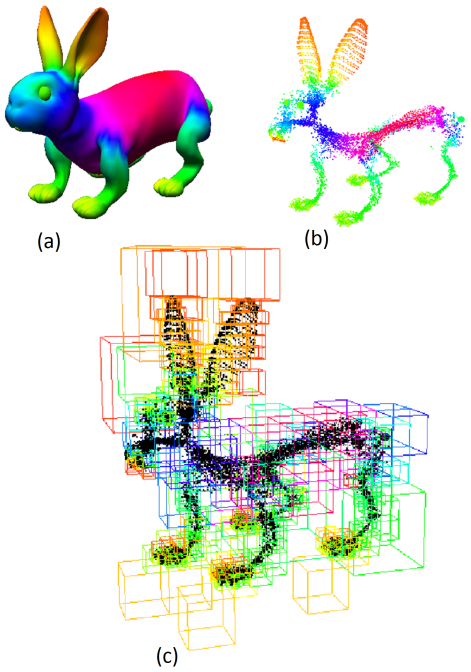


Figure 7: (a) mesh with SDF values, (b) projected points resembling medial axis, (c) octree nodes containing the average SDF values.

Smoothing in Texture: In our third approach we have chosen a more traditional method by smoothing the SDF values in texture. This requires that the mesh has a proper UV coordinates. In a case when the parametrization is missing, but we have a manifold model, it is possible to use Skeleton Texture Mapping [9] to create necessary UV coordinates. If performed on GPU, the fastest way is to use shaders. We bind our texture to a Frame Buffer Object (FBO), then we create a 2D orthogonal projection that has in the bottom-left corner coordinate (0,0) and in the top-right corner coordinate (1,1). Then we use OpenGL to draw the triangles into this texture using their UV coordinates and their SDF values as color. Afterwards we run our shader program that performs the smoothing operations. There are various methods that can be used, from Gaussian to bilateral, median or anisotropic filtering. After the smoothing is performed, we retrieve our results from texture. In a case when we end up with a vertex that has multiple UV coordinates, we can average the result. Result of Gaussian filtering can be seen in Figure 8.

Discussion: Smoothing on mesh: The parameter k is set manually depending on the number of triangles / vertices the mesh contains. We have tested the effect of various parameter settings on the consistency of the SDF on many

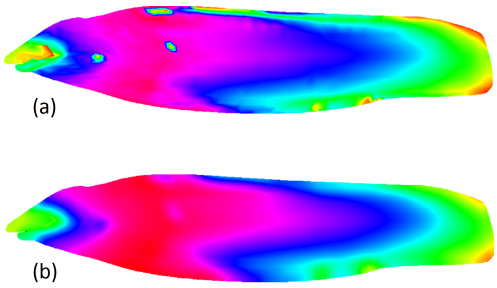


Figure 8: (a) before smoothing, (b) after smoothing.

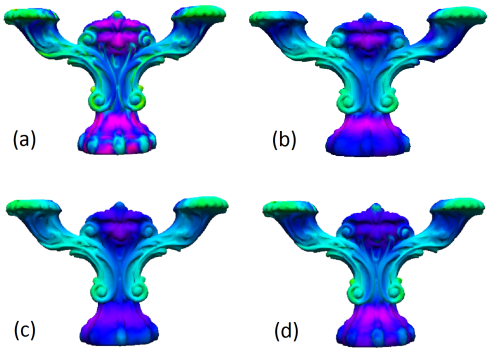


Figure 9: (a) no smoothing, (b) smoothing in texture, (c) smoothing on mesh, (d) smoothing on projected points.

meshes. In practice smaller meshes up to 20k triangles required radius of 2, while larger meshes like point cloud scans with 500k+ triangles radius of 5+. This smoothing approach gave best results when compared to others, however it was the slowest. It fails when the connectivity in mesh is not well defined or when the triangles does not have approximately the same sizes, in this case a lot of detail can be lost.

Smoothing on projected points: When creating an octree additional restrictions can be applied to further reduce the depth, while keeping the detail. In practice, the projected points are much closer to each other than the triangles in mesh and so we can use a smaller maximum depth, 8 was satisfying in most cases. Also minimum number of points for a node to branch can be increased to about 0.2% of all points. We can also compare the average value of the points against the octree dimensions and if it is bigger, then we do not branch. This method is faster than the first one, but the parallelization on GPU leads only to a slightly better results due to a lot of memory access. It also has an advantage that we do not have to set any radius value, because it is automatically acquired from the SDF values. Stronger smoothing can be obtained by iteratively running the method, which can even yield better results. It fails when the mesh surface is too irregular and the points are projected randomly, not forming a cloud like structure. Also it keeps the values at corners, which can be unwanted in some cases.

Smoothing in texture: This method is by far the fastest one, done in terms of ms even on large textures like 2048×2048. The need to have UV coordinates can be contraproductive because the automatic methods to create them are very slow. It can fail in multiple cases. First one is when the texture parametrization does not divide the mesh into logical parts. Second one is the texels of different parts are within the smoothing radius. Third one is when the projected triangles have different sizes. In practice we set the radius to 2 for a 256×256 texture and multiply as necessary, but it can vary depending on the model.

In Figure 9 the effect of all the methods on single model can be seen. Smoothing in texture performed similarly to smoothing on mesh, while smoothing on projected points kept more detail in corners.

5 Results

The testing was done on Intel Core i5, 2,67GHz with 4GB RAM and AMD Radeon R9 290, using 30 rays for each point. The algorithms were implemented in C++ in Visual Studio 2012 using standard OpenCL API.

Table 1 show basic performance of ray casting routine, outliers removal and smoothing. As for the smoothing chosen in the Table 1, we decided smoothing on projected points was the most suitable because it did not require any additional parameters. Table 2 shows octree creation. The time was measured on CPU. Table 3 shows smoothing on mesh using various k-ring area settings. The time was measured on CPU because we did not had a GPU implementation. Table 4 shows smoothing in a 2048×2048 texture using Gaussian Filter with various radius settings. Figure 10 shows percentage difference between GPU and CPU implementation when used on the same model with varying level of detail.

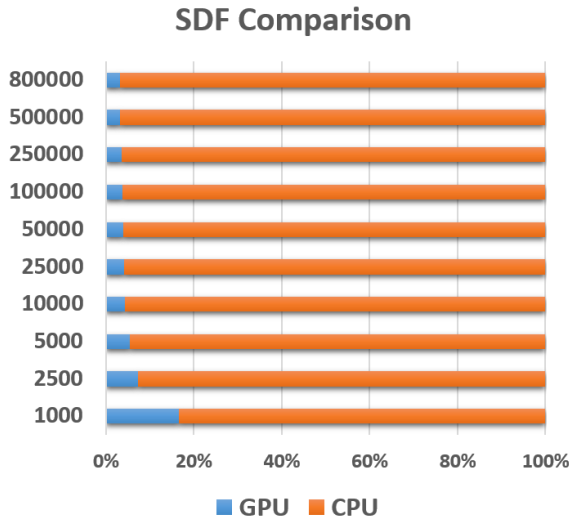


Figure 10: Benefits of GPU implementation with varying level of detail. Comparison was performed on model of Stanford Dragon.

Model	Faces	SDF Computation		Outliers Removal		Smoothing		Total	
Lizard	1 000 000	555,9	18,049	3,3	0.078	7,098	1,166	566,298	19,293
Stanford Dragon	500 000	234,1	8,721	1,32	0.047	3,385	0,576	238,805	9,344
Davy Jones' head	260 000	143,0	2,449	0,843	0,032	1,139	0,334	144,982	2,815
Skeleton	100 000	41,209	0,967	0,521	0,024	0,437	0,155	42,167	1,146
Buzz Lightyear	40 000	11,122	0,499	0,141	0,016	0,312	0,047	11,575	0,562
S-shape	20 000	4,274	0,219	0,063	0,015	0,171	0,016	4,508	0,250
Rabbit	15 000	3,026	0,156	0,047	0,008	0,109	0,015	3,182	0,179
Bottle	2 500	0,316	0,031	0,016	0,006	0,015	0,012	0,347	0,049
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU

Table 1: Results for both CPU and GPU computation of SDF. Listed times are in seconds. Total time does not include preprocessing.

Maximum Depth	Octree Creation
14	13,931
12	8,923
10	3,525
8	0,998
6	0,421
4	0,218
2	0,063

Table 2: Octree creation in preprocessing stage with varying maximum depth. Listed times are in seconds. We used model of Stanford Dragon with 500 000 triangles.

k-ring	Smoothing
8	26,567
7	20,015
6	14,882
5	10,764
4	7,566
3	5,024
2	3,120
1	1,762

Table 3: Smoothing on mesh using various k-ring areas. Listed times are in seconds. We used model of Stanford Dragon with 500 000 triangles.

6 Conclusion

In Section 3, we described the present state of methods used in the in the original paper [16]. In Section 4, we proposed our OpenCL implementation of the algorithm. We described all the steps necessary to perform the ray casting, outliers removal and smoothing on GPU. The various smoothing techniques which we subsequently developed (see Figure 9) can be used to increase robustness and overcome unwanted variations on mesh. Finally, in Section 5 we compared our GPU implementation against the CPU implementation and shown great speedup in terms of timing.

Radius	CPU smoothing	GPU smoothing
64	15,756	0,160
32	10,842	0,117
16	8,361	0,106
8	7,192	0,101
4	6,599	0,093
2	6,318	0,083

Table 4: Smoothing in a 2048×2048 texture using various radius settings. Listed times are in seconds. We used model of Stanford Dragon with 500 000 triangles.

Overall, in this paper we have proposed a parallel method for computing ray casting, outliers removal and smoothing steps of Shape Diameter Function that is performed on GPU using OpenCL. We have maintained the accuracy of the results while noticeably increasing it's speed.

References

[1] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[2] Andrea Baldacci. Gpu - accelerated shape diameter function filter for meshlab, 2011.

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

[4] Hyeong In Choi, Sung Woo Choi, and Hwan Pyo Moon. Moon: Mathematical theory of medial axis transform. *Pacific J. Math*, 1997.

[5] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical En-

gineering, Czech Technical University in Prague, November 2000.

[6] Maurizio Kovacic, Fabio Guggeri, Stefano Marras, and Riccardo Scateni. Fast Approximation of the Shape Diameter Function. *Proc. Workshop on Computer Graphics, Computer Vision and Mathematics (GraVisMa)*, Vol. 5, 2010.

[7] Peter Moller-Nielsen Kristof Rmisch. *Sparse Voxel Octree Ray Tracing on the GPU*. Ph.d. thesis, Department of Computer Science, Aarhus University, Denmark, September 2009.

[8] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 55–63, New York, NY, USA, 2010. ACM.

[9] Martin Madaras and Roman Ďurikovič. Skeleton texture mapping. In *Proceedings of the 28th Spring Conference on Computer Graphics*, SCCG '12, pages 121–127, New York, NY, USA, 2013. ACM.

[10] R. Marques, C. Bouville, M. Ribardire, L. P. Santos, and K. Bouatouch. Spherical fibonacci point sets for illumination integrals. *Computer Graphics Forum*, 32(8):134–143, 2013.

[11] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, October 1997.

[12] Khronos OpenCL and Aaftab Munshi. The opencl specification version: 1.0 document revision: 48, 2013.

[13] Daniel Schweri Philippe C.D. Robert. Gpu-based ray-triangle intersection testing. Technical report, Research Group on Computational Geometry and Graphics, Institute of Computer Science and Applied Mathematics, University of Bern, 2004.

[14] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, July 2002.

[15] Xavier Rolland-Nevière, Gwenaël Doërr, and Pierre Alliez. Robust diameter-based thickness estimation of 3d objects. *Graphical Models*, 75(6):279–296, 2013.

[16] Lior Shapira, Ariel Shamir, and Daniel Cohen-Or. Consistent mesh partitioning and skeletonisation using the shape diameter function. *Vis. Comput.*, 24:249–259, March 2008.

Deriving Shape Grammars on the GPU

Mark Dokter*
Supervised by: Markus Steinberger and Michael Kenzel†

Institute for Computer Graphics and Vision
Graz University of Technology
Graz / Austria

Abstract

Due to growing demand for computer generated graphical content, procedural modeling has become an important topic in the gaming and movie industry. Creating vast amounts of content by hand requires excessive amounts of manual labor. Using a procedural rule set, entire worlds can be generated by a computer. However, the traditional CPU-based derivation of a large city can take multiple hours, making rapid design iterations impossible. In this paper, we investigate different strategies to execute procedural modeling on graphics processors using CUDA. We compare a persistent threads megakernel approach to simple kernel calls and different rule queuing strategies. Along these lines, we explore the trade-off between precompiling an entire rule set and interpreting a rule set online.

Keywords: GPGPU, megakernel, procedural modeling, rule derivaton, shape grammar

1 Introduction

Generating graphical content in an automated fashion has become increasingly important during the last decade. Many recent computer games offer vast open virtual worlds, where the player can freely explore the environment. In the latest version of Grand Theft Auto for example, the area is not confined to a single city, but also includes its suburbs, where one can walk, drive or fly. Movies like Lord of the Rings show huge battlefield scenery with thousands of warriors fighting on wide open plains.

Those are examples of extensive use of digitally created content, which requires vast amounts of manual labor to produce. By automating content creation as much as possible, artists can spend more of their time on elements relevant to narrative and gameplay, rather than on creating peripheral scenery.

The task of creating reoccurring, parameterizable objects like houses or trees is well suited to procedural modeling. Rules for model production can be defined in a shape grammar. Starting from an initial set of shapes, these rules iteratively add detail to the scene. After fully evaluating such a rule set, the geometry which describes the entire scene is ready for rendering. However, grammar derivation for thousands of buildings can take many hours on a conventional CPU, even with several cores and a high clock rate.

One way to increase performance is parallelization. Parallelizing tasks and algorithms has gained much popularity with the introduction of general purpose GPU computing. With many cores on a single chip, the performance of a GPU is unmatched by any CPU, assuming a suitable parallelizable task. Procedural geometry generation is such a task, which can be, if done carefully, parallelized and computed efficiently on a GPU.

As will be discussed in the section on related work, various attempts of mapping this challenging task to a graphics processor have already been made. This work focuses on exploring the benefits and drawbacks of deriving precompiled rule sets versus interpreting them at runtime. Furthermore, various methods of controlling the GPU rule evaluation process will be subject to testing. This includes launching several successive kernels as well as deriving the entire scene using a single kernel launch, using a persistent threads megakernel approach.

2 Related Work

Currently, the most widely used grammar for procedural architecture modeling is CGA [12]. CGA is based on Stiny's work on *shape-grammars* [16] and *set-grammars* [18]. Furthermore, it uses split operations for facade modeling as proposed by Wonka et al. [19] and transformation operations similar to *L-systems* [13]. Approaches augmenting the functionality and usefulness of shape grammars exist on more general non-terminal symbols [3] and mesh refinement [2]. Apart from grammar based approaches to the procedural generation of geometry, other methods can be used to obtain high quality models [6, 7, 11].

Parallel grammar derivation has been investigated in various approaches which differ greatly in their strategy. Deriving L-systems on CPU clusters has been done by [20]. Considering the inherent parallelism of the algorithm, CPU clusters seem to be a good idea. However, when using a GPU, the results are already in the memory of the graphics card which is obviously more convenient for rendering.

*dokter@icg.tugraz.at
†steinberger@icg.tugraz.at, kenzel@icg.tugraz.at

A recent L-system generator for the GPU has been proposed by Lipp et al. [8]. In their work, they used multiple kernel launches to implement iterative rewriting of L-systems. Using a single thread per symbol without sorting the symbol stream has some drawbacks. First, memory accesses can become problematic if symbol sizes are not coherent. Second, thread divergence, which is the effect of threads taking distinct execution paths, results in different times the threads need to finish their work. On a GPU, where it is desirable to have as many threads occupied as possible at any point during run time, this effect causes some threads to wait on others which might take longer. This drastically impacts performance. And third, the management overhead for keeping track of where to store symbols quickly becomes a dominant factor. Thus, for context sensitive grammars, the derivation process was even slower on a GPU than on a CPU.

Shader based derivation of split grammars has been proposed and investigated by Lacz and Hart [4], Magdics et al. [9] and Marvie et al. [10]. The method by Lacz and Hart uses a render-to-texture loop and imposes the main workload of the algorithm on sorting intermediate symbols—similar to the overhead found in L-system generator by Lipp et al. The method Magdics et al. also requires several rendering passes. It tries to prevent divergence by using a different shader for each output symbol. In our evaluation, we incorporate an approach inspired by their work, efficiently grouping output symbols and launching individual kernels for each symbol type.

The approach by Magdics et al. avoids multi-pass rendering by using a fixed size stack. Using a fixed size stack has multiple drawbacks. First, recursion depth is limited. Second, stack elements might be spilled to slow global GPU memory. Third, parallelism is limited to the number of axioms. And fourth, divergence can play a crucial role, if objects do not have identical structure.

An approach focusing on parallelizing grammar derivation for procedural modeling of architecture has been published by Steinberger et al. [15]. The *PGA* grammar is based on CGA[12] and uses a software scheduling GPU framework [14]. To avoid divergence, their approach groups shapes, which are to be processed by the same rule. Additionally, they draw parallelism from the rule itself. PGA compiles the entire rule set to achieve high performance rule derivation. The work reported in this paper is a direct extension of PGA.

3 GPU Split Grammars

Split grammars, introduced by Wonka et al. [19], are specialized set grammars, which impose restrictions on the allowed shapes and operations to make the grammar simple enough for automated derivation, but sufficiently expressive to allow the modeling of many different objects.

A split grammar builds on the notion of shapes and set grammars. A *shape* can be defined as follows [17]:

Definition 3.1 *A shape is a limited arrangement of straight lines in three-dimensional Euclidean space.*

Split grammars operate on a set of *basic shapes*, which can have attributes, can be parameterized and labeled. These basic shapes form the core buildings blocks of split grammars. Examples for the geometry represented with basic shapes are boxes, spheres, cylinders, rectangles, etc. The parameters of these basic shapes define their extent, their position, etc. The label associated with the shape is often called symbol. This symbol can either be a terminal symbol $\in T$ or a non-terminal symbol $\in N$.

A *grammar* can be defined as a set of production rules R on a set of symbols U, using the following definition similar to the one given by Wonka et al. [19]:

Definition 3.2 *A grammar $G = (N, T, R, I)$ consists of the non-terminal symbols $N \subseteq U$, the terminal symbols $T \subseteq U$, a set of initial symbols (axioms) $I \subseteq N$ and a set of rewriting rules (productions) $R \subseteq U \times U^*$.*

A *rule* $a \longrightarrow B$ in a grammar is applicable to a non-terminal symbol $a \in N$, replacing it with B, whereas B can be any combination of non-terminals $\in N$ and terminals $\in T$.

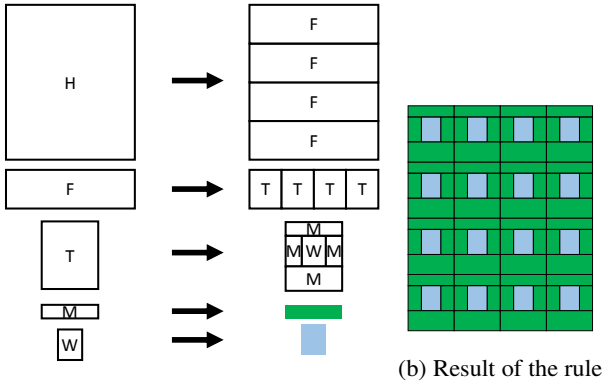
In a set grammar, the production process works on an active set of symbols. Initially, the active set consists of all axioms. During production, any non-terminal symbol from the active set of symbols is chosen and a fitting rule is executed on this symbol. The symbols generated by that rule are put back into the active set of symbols. This process continues, until there are only terminal symbols left in the active set.

In the case of split grammars, the production process works on shapes. Rules thus describe geometry operations on the input shape, generating any number of new shapes. For a grammar to be a split grammar, only two kinds of rules are allowed [19]:

- Split rules: A split rules splits a shape into multiple shapes, covering the exactly same volume as the input shape.
- Conversion rules: A conversion rules replaces a shape by zero to multiple shapes, where the generated shapes must be contained in the volume of the input shape.

These restrictions allow for a simple grammar derivation, as rules can only influence a constrained volume, as shown in Figure 1. Furthermore, every shape can be treated independently of the other shapes in the active set. This allows for a fully parallel production process. CGA, and consequently our grammar as well, do not have these restrictions and shapes can also increase in size, be moved or extruded. Furthermore, to simplify things, our implementation does not support control grammars lie CGA does.

To ease the process of writing rules, rules are usually composed of *operators*. Operators can be seen as basic geometric transformations executed in sequence to form a rule. Our grammar supports the transform-only operators



(a) A Simple Split Grammar (b) Result of the rule set (a)

Figure 1: A split grammar works a set of shapes, each associated with a symbol. Rules (a) replace one shape by a group of other shapes. Using split grammars, more complex objects can be generated from very simple rules.

Translate, Rotate, and Scale and generative transformations that produce more shapes than existed before the transformation. Those operators are Repeat and Subdivide. Furthermore, we support two operations that change the dimension of a shape: the Extrude operator, applied to a quad, generates a box the ComponentSplit operator, applied to a box, generates quads, representing the six faces of the box. Finally, we support the GenerateTerminal and the DiscardTerminal operator. The former calculates the geometry or simply copies the scaled model matrix (details in the implementation section 4). For instance, the third rule in Figure 1(a) splitting shape T into five shapes can be modeled as a combination of two Subdivide operators.

We use C++ template code to write define operators in the rule sets. Three example operators are described below with listings 1, 2 and 3 to illustrate the syntax:

- **Repeat** takes two parameters, a successive symbol and a shape and produces as many new shapes with the width specified by the second parameter as fit into the original shape. The operator can work on either of the other two dimensions.

Listing 1: Repeat Operator

```
1 repeat<X, 2, CallRule<Successor>>
```

applied to a box with width 8 will output four new boxes with a width of two (and the remaining extents according to the input shape), which all have the symbol ”Successor” as its successive symbol.

- **Subdivide** takes a varying amount of parameters and successive symbols plus the input shape. The first parameter is again the axis, the operation is applied to. The remaining parameters specify the relative width/height/depth for the newly generated shapes and their successive symbol, which can be different for each shape. The symbol can also be the same for every output shape, but has to be specified as many times as there are output shapes.

Listing 2: Subdivide Operator

```
1 subdivide<Y,
2           SubdivParam<500, CallRule<Successor1>,
3           SubdivParam<500, CallRule<Successor2>>>
```

Applied to a box with the height of four, Subdivide will produce two boxes with the height of two (and the remaining extents according to the input shape). The successive symbols of the two resulting boxes will be ”Successor1” and ”Successor2”, respectively.

- **ComponentSplit** takes an input shape and generates as many new shapes of lower dimension as are needed to represent the faces of the original shapes. Our implementation supports only the splitting of a box into six quads. The operator needs to be provided only with the six successive symbols (which may be all the same symbol, but in this case have to be specified six times).

Listing 3: Component Split Operator

```
1 Compsplit<CSP<CallRule<Bottom>,
2           CSP<CallRule<Top>,
3           CSP<CallRule<Right>,
4           CSP<CallRule<Left>,
5           CSP<CallRule<Back>,
6           CSP<CallRule<Front>>>>>>>
```

Using CUDA, a single thread can be launched for every shape in the active set, applying the rule associated with the shape’s symbol. Despite the great potential for parallel execution in split grammars, traditional GPU stream processing approaches are not well suited to fulfill the derivation process efficiently because work loads are highly irregular in split grammars, leading to thread divergence. Since our grammar descends from split grammars, this problem needs to be considered.

To avoid thread divergence, a scheduling system based on rule queuing can be set up to keep up the occupancy of a GPU [15]. The results of this rule scheduling paradigm are promising, as this system allows to generate whole cities in real time. However, this work only focuses on a single strategy to schedule rules: The entire GPU is occupied with a persistent threads approach [1]. Symbols of equal type are collected in queues, while workers draw elements from these queues. All rules have to be available at compile time, requiring a full recompile when altering the rule set. In this work, we investigate the alternative methods to schedule shape grammars on the GPU. On the one hand, we investigate the benefits and downsides of scheduling whole rules versus scheduling work for each operator individually. On the other hand, we investigate the trade-off between compiling entire rule sets and interpreting the provided rule set during runtime.

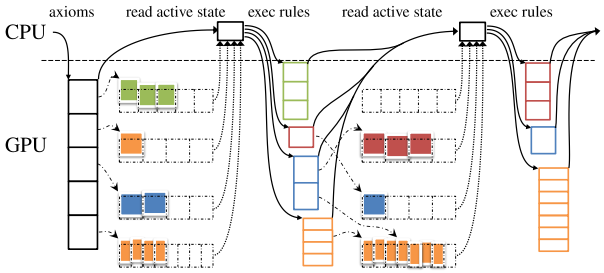


Figure 2: Using an individual queue for each rule, we provide an iterative shape rewriting algorithm, which does not suffer from divergence. At first all axioms are being placed in the queues. Then, we read the queue fill rate back to the CPU before launching just enough threads to process all queued shapes. We continue this process until there are only terminal shapes left.

3.1 Iterative Production

The most straight forward way to tackle shape grammar evaluation is starting a single thread for each symbol in the current active set. This is similar to the approach by Lipp et al. [8]. As mentioned before, this method has the drawback of extensive thread divergence. Inspired by the approach by Laine et al. [5], we avoid thread divergence, providing individual queues for each rule. Before running the rule evaluation, we allocate a queue for each rule on the GPU. We then insert the axioms into the per-rule queues. After querying the queue fill rates, we start an individual kernel for each queue, launching just as many threads as there are elements in each queue. Each thread then fetches one element from the queue and executes the rule associated with it. During rule evaluation, new shapes are generated, which are again inserted into the respective queues. Terminal shapes are placed into a separate set of arrays, for which no rule evaluation is taking place. These arrays are later used for rendering. After all kernel launches are completed, we read the queue fill rates from the GPU and again launch kernels to evaluate rules for all shapes currently being held by all queues. We continue this process until all non-terminal shapes have been processed, *i. e.*, all queues reach an empty state. Shapes currently being queued represent the current active set. This process is visualized in Figure 2.

Using this approach, all threads within one kernel evaluate the same rule, executing the same set of instructions. Thus, no thread divergence occurs and execution is efficient on the GPU hardware. While this approach is set up easily, deriving a whole rule set requires many kernel launches. Additionally, the queue fill rates need to be read back from the GPU before a new set of kernels can be launched. This step cannot be avoided, as the number of threads to be launched needs to be known.

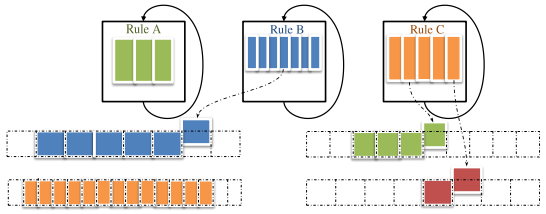


Figure 3: As alternative rule derivation algorithm, we use a persistent megakernel setup. Worker blocks are running in an endless loop. At the beginning of each loop iteration, they draw a new setup of shapes from one of the queues and evaluate the associated rules, before inserting the generated shapes back into the queues. The kernel is kept alive until all non-terminal shapes have been processed.

3.2 Persistent Megakernel Production

As alternative way to tackle shape grammar evaluation on the GPU, we use a persistent threads approach [1]. We again use a single queue per rule. But instead of launching a new kernel for every rule production, we run threads in an endless loop. In every loop iteration, each thread draws a shape from one of the queues and executes its associated rule. If new shapes are being generated, we add them back into the respective queues. As shapes are being drawn from the queues and inserted into the queues concurrently, we use a flag per queue element to avoid errors due to read-before-write dependencies [14]. All threads continue in their loop, until all queues are empty and no thread is still evaluating a rule. To avoid thread divergence, we force all threads within a block to draw shapes from the same queue in every iteration. This setup is outlined in Figure 3.

A persistent megakernel setup avoids synchronization with the CPU and does not have any kernel launch overhead. On the downside, all rules must be compiled into the same kernel. As kernels are optimized as a whole, the characteristics of the most resource-hungry rule determines the efficiency of all others. Furthermore, the persistent setup requires a more complex queuing strategy to avoid errors due to read-before-write dependencies. As our grammar does not introduce any priority among rules, shapes can be drawn from any queue at the beginning of each iteration. To avoid idle threads, we circle through all queues in a round robin fashion and only draw shapes from a queue if there are enough shapes in the queue to provide all threads in the block with work.

3.3 Precompiled Rules

The goal of the precompiled rule set approach is to leave as many decisions as possible to the compiler. For this approach, we require the complete rule set to be specified beforehand. This includes all rules, their parameters, and outputs. The only information not required in advance are the axioms. All computations and branch decisions that are not input dependent need only be done once, so we perform

them at compile time. Thus, during runtime, all operators can be executed without any additional information. No lookups to rule tables or symbol translations are needed.

The anticipated result is that this method achieves the best possible performance when compared to approaches that can adjust their behavior to different rule sets during runtime. Precompiled rules lose the flexibility of changing the rule set at run time and need significantly longer compile time. Usually, the performance gain from precompiling a rule set would be leveraged in production systems, such as games, once the design phase is finished and no interactivity is needed anymore.

Precompiled rule sets are evaluated in a ”one rule at a time” fashion by our software. This means that several operators can be chained together in a rule which forms the procedure to be called by the scheduler. While this approach has low scheduling overhead, it may not exploit all options for parallelism. The same operators are likely to be used in different rules and could be executed efficiently in parallel. However, the scheduler only knows about rules, thus it treats all rules as different. Moreover, the complexity of such a precompiled rule set can increase quickly. This circumstance not only imposes high requirements on the quality of the compiler, but also, if not implemented carefully, results in very high compile times, which may only be tolerable for production use.

3.4 Interpreted Rules

Interpreting rules at runtime gives flexibility when designing new objects at the cost of performance. With this approach, rule sets can be imported from file or created interactively, possibly with a rule editing tool—ideally with a graphical user interface.

In the interpreted mode, our solution evaluates rule sets in a ”one operator at a time” fashion. This means that every rule is broken apart into its operators and intermediate shapes are generated. These shapes are handed over to the scheduler. To determine how operators are strung together to rules for the currently used rule sets, we generate a dispatch table. This table holds for each (intermediate) symbol the operator to be executed, the parameters for each operator and all generated output symbols. During runtime, whenever a thread starts the evaluation for a certain shape, it fetches all parameters from the dispatch table and executes the requested operations. To avoid divergence, we keep one queue per operator. When a new shape is generated, we look up which operator should be called for it next and insert it into the respective queue.

The major advantage of interpreted rule sets is the ability to alter the rule set during run time, allowing for efficient prototyping and immediate feedback. Another advantage of interpreted rules is that the scheduler is now exposed to all available parallelism: shapes to be executed by the same operator can be grouped, even if they are used in completely unrelated rules.

4 Implementation

Our implementation is written in CUDA and C++ and makes heavy use of templates. Rendering is done in two different ways using OpenGL. The two variants are instanced and non-instanced rendering. In the non-instanced method, vertex, normal and instance data is generated by the terminal operator. The instanced method renders basic shapes (boxes, quads, etc) and only needs to apply the calculated model matrix to put the shape into its place in the final rendering.

Using instanced rendering has three advantages: First, during terminal evaluation, less data has to be written to slow global GPU memory, as only the matrices need to be copied. Second, less storage is required between generation and rendering. And third, during rendering, less data needs to be read, saving memory bandwidth. However, the number of vertices of the basic shapes is too low for an efficient usage of instanced rendering. Thus, rendering is actually slower using instancing.

To implement a shape, we store its type, size and the model matrix (and the symbol ID in the interpreted method). All operators, except the GenerateTerminal operator, only alter these attributes, which is all the information needed to produce the geometry data. Using the non-instanced rendering method, GenerateTerminal calculates, according to the type of the shape, the vertex attributes and stores them in an OpenGL buffer, which is mapped to CUDA before the generation process starts. If we use instanced rendering, all that is left to do for the GenerateTerminal operator, is to store the model matrix in the OpenGL buffer.

The precompiled method is implemented using the template meta-programming paradigm. All rule sequence decisions are made by the compiler according to the rule definitions, which creates instances of rules and operator chains at compile time. The rule definitions are written in C++ template code as shown in listing 4. We use the template code not only to generate the operator chains for the precompiled method, but also to fill the dispatch tables for the interpreted case. However, the compile process in the interpreted case does not involve the generation of GPU code, only the CPU code generating the dispatch table. Thus, a full runtime adjustment of the rule set could easily be achieved using a custom parser.

Listing 4: Sample Rule Set

```
struct RuleB : RuleT<Box, IfSizeLess<X, 200,
            DiscardTerminal, GenerateTerminal> > {};

struct RuleA : RuleT<Box,
         translate<0, 567, 0, GenerateTerminal > > {};

struct StartRule : RuleT<Box,
           rotate<45000, 45000, 0, subdivide<X,
           SubdivParam<270, CallRule<RuleA>,
           SubdivParam<160, CallRule<RuleB>,
           SubdivParam<300, CallRule<RuleA>,
           SubdivParam<270, CallRule<RuleB>
           > > > > > > > {};

typedef RuleSet<RS<StartRule, RS<RuleA,
          RS<RuleB> > > > TheRules;
```

	Houses	S. Sierpinski	M. Sierpinski
Rules	9	5	15
Operators	11	4	4
Terminals	332.8k	3.20M	1.35M
Vertices	7.99M	75.8M	32.5M
Indices	11.99M	115.2M	48.7M

Table 1: Test scene statistics

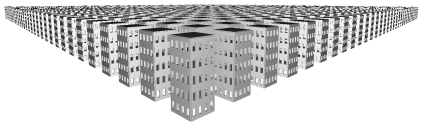


Figure 4: The *Houses* testcase shows a scene of 32×32 simple house models.

5 Results

As this paper focuses on the rule derivation process, only rudimentary shaders have been applied to visualize the produced geometry. The absence of visually appealing rendering, as well as other features not relevant for this paper, like optimizing the number of objects that need to be generated or ignoring geometry that need not be regenerated for every frame is the topic of future work. The results of the tests presented in this section were carried out on a system with an Intel Core i7-3770 CPU at 3.4 GHz, 16 GB of main memory and a NVIDIA Geforce GTX TITAN with 6 GB VRAM.

We compare eight different configurations, which are variations of interpretation and precompilation, instanced and non-instanced rendering, as well as iterative production and persistent megakernel production. We applied these variation to three different rule sets: *Houses*, *Single Sierpinski*, and *Multi Sierpinski*. The statistics for these three rule sets are summarized in Table 1. Example views for all scenes are shown in Figure 4-6.

The evaluation results are shown in Table 2-4. In all examples, the generation time in the instanced variant was between three to five times lower than the non-instanced variant. The highest difference was achieved in the Multi Sierpinski test case, which generates a vast amount of geometry with relatively few rule evaluations per terminal. This fact is clearly visible when looking at the number of

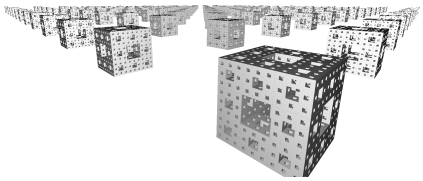


Figure 5: *Multi Sierpinski* consists of 13×13 Sierpinski Cubes at recursion depth 3.

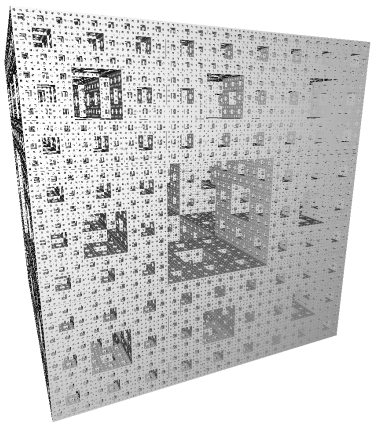


Figure 6: *Single Sierpinski* shows one deep Sierpinski Cube at recursion depth 5.

			t_g	t_r	load	store
int	n-inst	IP	16.6	2.4	1.55	28.51
		PMK	15.3	2.3	1.50	26.46
	inst	IP	4.5	5.7	1.33	7.72
		PMK	6.6	5.7	1.30	8.97
pre	n-inst	IP	21.4	2.4	1.11	32.93
		PMK	11.9	2.4	0.75	22.56
	inst	IP	2.8	5.7	0.75	4.72
		PMK	2.5	5.8	0.64	5.40

Table 2: Evaluation results for Houses, including interpreted (int) and precompiled (pre) rule sets; non-instanced (n-inst) and instanced (inst) rendering, as well as iterative production (IP) and persistent megakernel production (PMK). Generation time (t_g in ms) corresponds to the time needed for rule evaluation, t_r is the time spent in OpenGL rendering (ms), and load/store correspond to DRAM load and store requests to global GPU memory during grammar derivation.

			t_g	t_r	load	store
int	n-inst	IP	144.5	22.3	14.9	239.4
		PMK	103.3	22.1	11.7	188.1
	inst	IP	33.4	55.3	13.1	51.9
		PMK	32.3	56.0	11.6	52.0
pre	n-inst	IP	196.4	22.5	9.0	306.3
		PMK	105.7	22.0	5.8	190.4
	inst	IP	17.9	55.0	6.2	33.3
		PMK	21.8	55.3	5.8	34.7

Table 3: Evaluation results for Single Sierpinski

			t_g	t_r	load	store
int	n-inst	IP	57.6	9.2	4.6	98.2
		PMK	45.8	9.1	3.9	84.6
	inst	IP	10.7	23.2	3.8	18.6
		PMK	9.4	23.5	3.7	17.7
pre	n-inst	IP	84.5	9.3	2.3	131.1
		PMK	52.0	9.2	2.7	95.5
	inst	IP	5.5	23.3	1.1	11.3
		PMK	6.6	23.2	1.2	13.5

Table 4: Evaluation results for Multi Sierpinski

DRAM stores. The rendering time itself was hardly influenced by instancing, confirming that rendering really simple shapes using instancing is not more efficient than rendering the uncompressed geometry.

Surprisingly, in the non-instanced variant, interpreted rule evaluation was faster than precompiled in half of the cases, achieving an overall faster average rule evaluation by 12%. In the instanced variant the relationships are reversed, with precompiled outperforming interpreted by 84% on average. In all instanced tests, percompiled could generate the geometry faster. These are very interesting results, as precompiled is only significantly faster, when there is less memory traffic involved, due to the use of instancing. We can only assume that the interpreted evaluation can catch up in the non-instanced variant because all terminal operators are collected in the same queue. Thus, the terminal generation itself is highly efficient in comparison to the precompiled rule derivation, where the terminal generation is mixed with other operations. Thus, the interpreted version generates more homogeneous memory access patterns and overall runs faster. This is also reflected by the lower number of DRAM stores in the interpreted non-instanced versions when comparing interpreted to percompiled. In the instances variants, these numbers are reversed. Most of the memory access of the interpreted evaluation is due to dispatch table lookups and intermediate symbol generation, thus slowing down the generation process.

When comparing our iterative production implementation against the persistent megakernel approach, one can observe that the persistent megakernel implementation is on average 20% faster than the iterative production. In nine of the twelve cases, persistent megakernels were faster. Interestingly, there is no generalizable pattern visible, as to when iterative production works better. In the Houses test case, iterative production gives the best results for interpreted+instanced, for Single and Multi Sierpinski it achieves the best performance for precompiled+instanced.

Overall, we can observe that persistent megakernel production seems to work faster on average than iterative production. Instancing always increases performance. If instancing is used, precompiled rule sets are better than interpreted rule sets. If instancing is not used, terminal generation dominates performance, for which the interpreted rule sets are slightly faster, as they are able to merge the terminal operators.

When looking at the raw generation times, we can observe that the fastest method can generate 135 million terminals per second (MTPS) in the Houses test case, 178 MTPS in the Single Sierpinski test case and 247 MTPS for the multi Sierpinski rule set.

6 Future Work

Since the focus of this paper is the evaluation of different rule scheduling strategies, we omitted the implementation of features which only affect appearance and not performance. These features include textured rendering, auxiliary scenery like roads, water, vegetation and varying elevation of the ground. Also the support of imported off-line generated models would make the scene more lively. Furthermore, a randomization of input parameters, so the generated shapes do not all look alike, would be essential for producing realistic scenes. For the testing setup of our implementation, the use of boxes and quads was sufficient. To build more realistic housing procedurally, many more shapes could be implemented, like cylinders, cones and wedges. We plan to add these features in the future.

A rule editor to specify interpreted rules at run time would be beneficial to the usability of our solution, as writing rules off-line is not very intuitive, especially for generating complex models. Such an editor would ideally support writing rules in an already established shape grammar and could even support using a rule database, so users can import and export model descriptions like it is already done for conventional 3D models.

An interesting feature to implement is proper use of instanced rendering. While in our case it was enough to render instances of basic shapes to prove that instanced rendering is desirable when constantly generating geometry every frame, to save bandwidth, the vertex data for basic shapes is far to low to justify the instancing overhead for the rendering alone. Rendering instances of fully generated objects with a reasonable amount of vertex data would use the full potential of instanced rendering.

Last but not least, an important aspect of CGA is context sensitivity. In our implementation this was deliberately left for future investigation, since the complex matter of rule interdependency is out of scope of this work.

7 Conclusion

We have shown in this work that scheduling of rule derivation work load on a GPU in the context of grammar based procedural modeling has several aspects influencing performance that have to be considered carefully. First, decision making can be offloaded to the compilation stage in order to avoid expensive branching at run time.

Second, the proper utilization of GPU programming paradigms, while being partly platform dependent, as we focus primarily on NVIDIA CUDA technology, is essential in order to avoid wasting precious resources, slow memory accesses and thread divergence.

Third, the amount of data being moved when generating geometry on the fly ought not to be underestimated, which is why the use of instanced rendering is the preferred method to render massive amounts of procedurally generated models.

Furthermore, when applying excessive template programming, the quality of a decent compiler is not to be underestimated, as is the consideration how code generation (especially its memory consumption) will respond to chaining templates together recursively.

References

[1] Timo Aila and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. High Performance Graphics*, pages 145–149. ACM, 2009.

[2] Sven Havemann. *Generative Mesh Modeling*. PhD thesis, TU Braunschweig, 2005.

[3] Lars Krecklau, Darko Pavic, and Leif Kobbelt. Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Comp. Graph. Forum*, 29:2291–2303, 2011.

[4] Patrick Lacz and John C. Hart. Procedural Geometry Synthesis on the GPU. In *Workshop on General Purpose Computing on Graphics Processors*, pages 23–23, 2004.

[5] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 137–143, New York, NY, USA, 2013. ACM.

[6] Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. By-example synthesis of architectural textures. *ACM Trans. Graph.*, 29:A84, 2010.

[7] Jinjie Lin, Daniel Cohen-Or, Hao Zhang, Cheng Liang, Andrei Sharf, Oliver Deussen, and Baoquan Chen. Structure-preserving retargeting of irregular 3D architecture. *ACM Trans. Graph.*, 30(6):A183, December 2011.

[8] Markus Lipp, Peter Wonka, and Michael Wimmer. Parallel Generation of Multiple L-systems. *Computer and Graphics*, 34(5), 2010.

[9] Milán Magdics. Real-time generation of l-system scene models for rendering and interaction. In *Proceedings of the 25th Spring Conference on Computer Graphics*, SCCG '09, pages 67–74, New York, NY, USA, 2009. ACM.

[10] Jean-Eudes Marvie, Cyprien Buron, Pascal Gautron, Patrice Hirtzlin, and Gaël Sourimant. GPU Shape Grammars. *Comp. Graph. Forum*, 31(7-1):2087–2095, 2012.

[11] Paul Merrell and Dinesh Manocha. Model Synthesis: A General Procedural Modeling Algorithm. *Visualization and Computer Graphics, IEEE Trans.*, 17(6):715–728, 2011.

[12] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural Modeling of Buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006.

[13] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990.

[14] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: Dynamic Scheduling on GPUs. *ACM Trans. Graph.*, 31(6):A161, 2012.

[15] Markus Steinberger, Michael Kenzel, Bernhard Kainz, Jörg Müller, Peter Wonka, and Dieter Schmalstieg. Parallel generation of architecture on the GPU. *Comp. Graph. Forum*, 33, 2014.

[16] George Stiny. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, Basel, 1975.

[17] George Stiny. Introduction to shape and shape grammars. *Environment and planning B*, 7(3):343–351, 1980.

[18] George Stiny. Spatial Relations and Grammars. *Environment and Planning B*, 9:313–314, 1982.

[19] Peter Wonka, Michael Wimmer, François X. Sillion, and William Ribarsky. Instant Architecture. *ACM Trans. Graph.*, 22(3):669–677, 2003.

[20] Tingjun Yang, Zhengge Huang, Xingsheng Lin, Jianjun Chen, and andJun Ni. A parallel algorithm for binary-tree-based string re-writing in the l-system. In *Proceedings of the Second International Multi-Symposiums on Computer and Computational Sciences*, IMSCCS '07, pages 245–252, Washington, DC, USA, 2007. IEEE Computer Society.

Computer Vision

Fully Automated Real-Time Vehicles Detection and Tracking with Lanes Analysis

Jakub Sochor*
Supervised by: Adam Herout†

Faculty of Information Technology
Brno University of Technology
Brno / Czech Republic

Abstract

This paper presents a fully automated system for traffic surveillance which is able to count passing cars, determine their direction, and the lane which they are taking. The system works without any manual input whatsoever and it is able to automatically calibrate the camera by detecting vanishing points in the video sequence. The proposed system is able to work in real time and therefore it is ready for deployment in real traffic surveillance applications. The system uses motion detection and tracking with the Kalman filter. The lane detection is based on clustering of trajectories of vehicles. The main contribution is a set of filters which a track has to pass in order to be treated as a vehicle and the full automation of the system.

Keywords: motion detection, tracking, vehicles, traffic surveillance camera, direction detection, lanes detection, real-time

1 Introduction

This paper presents a fully automated system for traffic analysis. These types of analysis systems have a wide spectrum of usage. For example, it is possible to monitor the traffic or try to predict characteristics of the future traffic flow. The presented system is able to count passing cars, determine their direction and lane which they are taking. The goal is to run the system without any manual calibration or input whatsoever. The full automatism of the system is required if the system should be usable with already mounted uncalibrated cameras which are spread over highways. Therefore, the camera is automatically calibrated prior to running the traffic surveillance system. Real time processing is another requirement which needs to be satisfied for usage in real traffic surveillance applications.

Some methods for calibration of the camera require user input [29, 3] and therefore they can not be used in fully automated systems. Approaches for the calibration are usually focused on detection of vanishing point of the direction parallel to moving vehicles [6, 10, 23, 25]. There are several ways how to detect the vanishing point. Detected lines [25, 6] or lanes [25, 10] can be used for obtaining this vanishing point. On the other hand, Schoepflin and Dailey [23] use motion of vehicles and assume that they have straight parallel trajectories. Kanhere et al. [16] detect vehicles by a boosted detector and observe their movement, and Zhang et al. [30] analyze edges present on the vehicles.

Figure 1: Example of video scene processed by the proposed traffic analysis system. Information about passing cars and their directions are displayed in output video.

A popular approach to detection and tracking of vehicles is to use some form of background subtraction and Kalman filter [15] to track the vehicles [12, 21, 14, 28, 1, 4, 7, 20, 17, 22]. Other approaches are based mainly on detection of corner features, their tracking and grouping [2, 13, 5]. Also, Cheng and Hsu [4] use pairing of headlights for the detection of vehicles at night.

Two main approaches are used for the detection of lanes. The first one is based on detection of the lane dividing lines [13, 18]. The other approach is based on motion of vehicles and their trajectories. Tseng et al. [28] use a virtual line perpendicular to vehicles' motion and compute intersections of the line with trajectories of vehicles. Hsieh et al. [12] use a two-dimensional histogram of accumulated centers of vehicles and Melo et al. [20] approximate the trajectories with low-degree polynomials

*xsocho06@stud.fit.vutbr.cz
†herout@fit.vutbr.cz

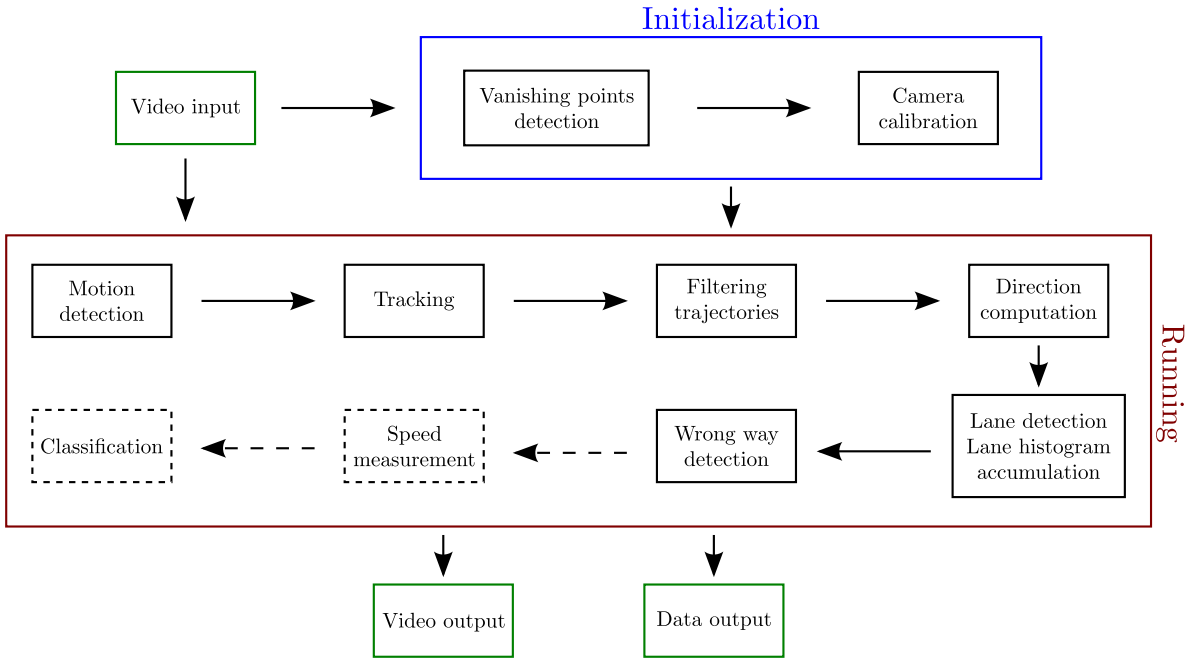


Figure 2: Pipeline of processing of the input video stream. Parts of the pipeline which will be implemented in the future, namely Classification and Speed measurement, are shown in dashed boxes.

and cluster these approximations.

The system proposed in this paper uses detection of vehicles by background subtraction [26, 31] and Kalman filter [15] for tracking. Prior to running the algorithm, the camera is calibrated by the detected vanishing points and the vanishing point of direction parallel to the motion of vehicles is used for higher accuracy of tracking. The detection of lanes is based on trajectories of vehicles and their approximation by a line.

2 Proposed Method for Traffic Surveillance

This section of the paper presents methods used in the system for detection and tracking of cars. The direction and lane detection is also discussed in detail. The overall processing pipeline is shown in Figure 2.

The main goal of the system is to create statistics of traffic on a road which is monitored by a camera. These statistics include the number of passed cars, their direction and lane.

2.1 Initialization

It is required to initialize the system prior to processing a video stream. The main purpose of the initialization is to find vanishing points of the scene and use the vanishing points to calibrate the camera. This is performed in a fully automated way and no user input is used. Arrows directed to the vanishing points are used for visualisation of the vanishing points. An example of the visualisation of the vanishing points is in Figure 3.

The vanishing point of the direction parallel to the vehicle movement is denoted as the first vanishing point. The second vanishing point has perpendicular direction to the movement of vehicles and the third vanishing point is perpendicular to the ground plane. However, only the first vanishing point is required for the tasks described in this paper; therefore, only detection of this vanishing point will be described. The detection of the other vanishing points is described in a paper written by Dubská et al. [8], currently submitted to IEEE Transactions on Intelligent Transportation Systems.

First Vanishing Point Detection

Corner feature tracking is used for the detection of the first vanishing point. Hence, Good Features to Track [24] are detected in the video stream and KLT tracker [27] is used for the tracking of the corner features. Detected motion of the tracked features is extended into a line which is defined by image points (x_t, y_t) and (x_{t+1}, y_{t+1}) which are positions of the feature in frame t and $t+1$.

All these lines are accumulated into the *diamond space* [9] until the initialization is terminated. The initialization is terminated when the global maximum of the diamond space is bigger then a predefined threshold and therefore a sufficient number of lines was accumulated. Afterwards, the coordinates of the global maximum in the diamond space are transformed into coordinates of the vanishing point in the image plane.

The diamond space is based on Cascaded Hough trans-

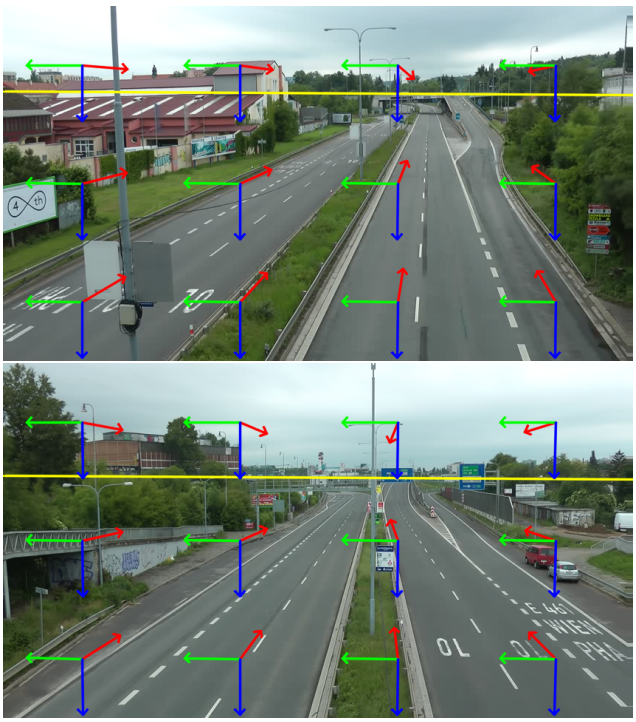


Figure 3: Detected vanishing points. Red arrows are pointing to the first vanishing point, green to the second one, and the third vanishing point is defined by the blue arrows. Yellow horizon line connects the first and second vanishing point.

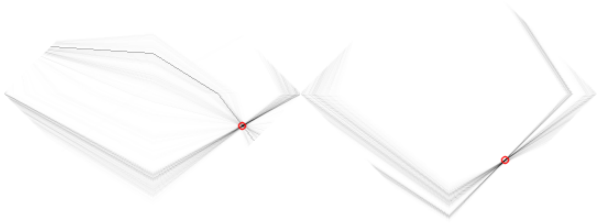


Figure 4: Examples of diamond spaces for detection of the first vanishing point with located global maximum

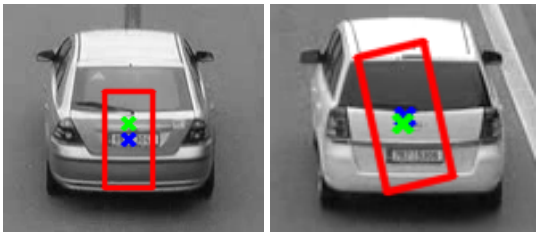


Figure 5: Examples of matching rectangles (red) for predicted object location (blue). The actual center of the detected connected component is drawn by green color. The figure shows that the longer side of the rectangle is directed to the first vanishing point.

form and parallel coordinates. Each line which is accumulated into the diamond space has to be transformed into coordinates in this space. The transformation divides the line into three line segments which are accumulated into the diamond space. Examples of the diamond space are in Figure 4.

It should be noted that the system uses a video downsampled to a framerate close to 10 FPS, so that the movement of corner features is detectable and measurable.

2.2 Vehicle Detection and Tracking

The vehicle detection is based on motion detection in the video scene. Mixture of Gaussians background subtraction [26, 31] is used for the motion detection. Also, shadow removal [11] is used for higher accuracy of the motion detection. Noise in the detected motion is removed by morphological opening followed by morphological closing. Detected connected components are considered to be a potential vehicle. The motion detection approach was selected mainly for its speed.

Kalman filter [15] is used for prediction of the new position of a car and for associating cars in consequent frames. The state variable of the Kalman filter $(x, y, v_x, v_y)^T$ contains the current position of the car and its velocity in image coordinates.

Several conditions are used for matching an object in the consequent frame to its predicted location. The first condition states that the matched object must have similar colors. This condition is enforced by correlating histograms of objects in HSV color space. The second and last condition is that the center of matched object must be inside of so called *matching rectangle*. The predicted location of a car is the center of this matching rectangle and the longer side of the rectangle is directed towards the first vanishing point, as it is shown in Figure 5, and the matching rectangle has size 30×15 pixels. This condition is built on the assumption that the vehicle is going either in the direction towards the vanishing point or from the vanishing point, and therefore it is expected that in this direction can be higher displacement from the predicted location. Lastly, the closest connected component which meets the conditions presented above is found for each object and its predicted location in the consequent frame.

When a match is not found in several consequent frames, the tracked object is removed from the pool of tracked objects. Several filters are used for determining if the object should be accounted in the statistics of passed cars. The trajectory of the object is approximated by a line using least squares approximation. After that, the distance of the first vanishing point from the line is measured. Let us denote this distance as d_{vp}. Also, the ratio r, Eq. (1), between passed distance and maximal possible distance which an object can pass in the given trajectory is measured, Figure 6 shows the positions of P_e, P_s, L_e and L_s. The object is accounted in the statistics as a vehicle if the

acc variable is equal to 1, Equation (2), where t_{vp} and t_r are predefined thresholds.

$$r = \frac{||P_e - P_s||}{||L_e - L_s||} \quad (1)$$

$$acc = \begin{cases} 1 & d_{vp} \leq t_{vp} \text{ and } r \geq t_r \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

2.3 Direction Estimation and Lane Detection

For a new vehicle which is about to be added to the statistics, the direction of the vehicle and its lane is calculated. Rule (3), which compares the relative positions of the first vanishing point and the last and first position of the vehicle, is used for computing the direction.

$$dir = \begin{cases} \text{To VP} & ||VP_1 - P_e|| < ||VP_1 - P_s|| \\ \text{From VP} & \text{otherwise} \end{cases} \quad (3)$$

The detection of lanes is based on clustering of the trajectories of cars. Therefore, the trajectory is also approximated by a line with least squares approximation, see green line in Figure 6. Each cluster of the lines corresponds to a lane in the monitored traffic surveillance scene and the clustering is performed in a one-dimensional space, where the values of the trajectory lines are their angles with axis x in the image. The clusters itself are searched as local maxima in the histogram of the angles. Hence, the clusters have to be a local maximum in the histogram in a predefined surroundings and also the maximum has to have at least a predefined amount of accumulated lines. The closest lane is assigned to a new passing vehicle as the lane which the vehicle is using. The closest lane computation is also based on the angles of the trajectory line and the lane with axis x.

This clustering is always performed after every 200 trajectory lines are accumulated and a unique identification number is assigned to each cluster. Let us denote the set of clusters as $C_N = \{(c_1, a_1), \ldots, (c_n, a_n)\}$ where N is the number of accumulated lines, and pair (c_i, a_i) denotes one cluster, where c_i is its identification number and a_i the angle corresponding to the found local maximum. Correspondences for clusters C_N and C_{N-200} are searched in order to obtain the temporal consistency of detected lanes in the scene. The clusters' identification numbers would change after every 200 accumulated lines if the correspondences were not found; and therefore, it would be impossible to create long-term statistics for cars passing in the detected lanes.

The identification number of the found correspondence is assigned to a cluster if the correspondence is found. A new unique identification number is assigned to the cluster otherwise. The correspondence for a cluster $(c_i, a_i) \in C_N$ is a cluster $(c_j, a_j) \in C_{N-200}$ for which (4) and (5) hold. The distance function is computed according to Equation (6) which compensates that the angles 0 and 2π have distance from each other 0.

$$a_j = C_{N-200}\left(\arg\min_c |dist(C_{N-200}(c), a_i)|\right) \quad (4)$$

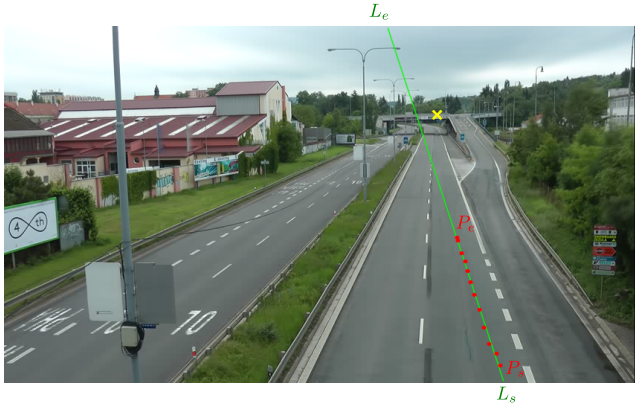


Figure 6: Measured distances for a passed object. The distance between approximated line (green) and the first vanishing point (yellow) is measured. Also, the distance between the first and last (P_s, P_e) point of the track of a vehicle is measured. The maximal distance which is possible to pass with a given trajectory is also measured (distance of L_s and L_e).

$$dist(a_j, a_i) \leq t_d \quad (5)$$

$$dist(x, y) = \min(2\pi - |x - y|, |x - y|) \quad (6)$$

The dominant direction is also computed for each cluster c of the trajectory lines. The dominant direction dir_c is computed according to (7), where l_{VP} is the amount of the trajectories in the cluster which have direction towards the first vanishing point and l is the number of all trajectories in the cluster. Reasonable value for threshold t_{dom} is 0.1.

$$dir_c = \begin{cases} \text{To VP} & \frac{l_{VP}}{l} \geq 1 - t_{dom} \\ \text{From VP} & \frac{l_{VP}}{l} \leq t_{dom} \\ \text{Mixed} & \text{otherwise} \end{cases} \quad (7)$$

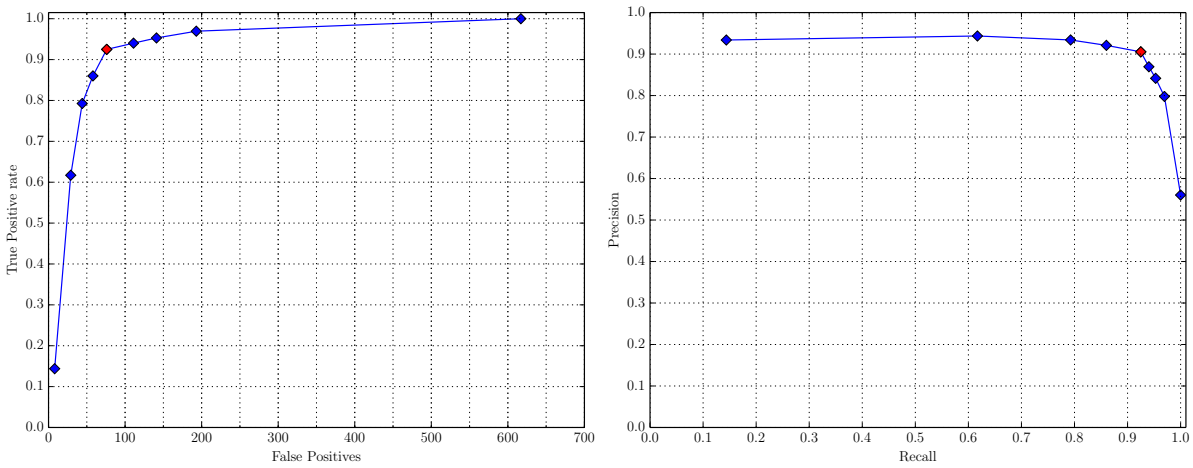
When the dominant direction for a lane is known, it is possible to detect vehicles which are traveling in wrong way. The detection is based on the detected direction *dir* of the vehicle and the dominant direction dir_c of the lane which the vehicle belongs to. The wrong way variable *ww* is determined by (8).

$$ww = \begin{cases} \text{True} & dir = \text{To VP} \wedge dir_c = \text{From VP} \\ \text{True} & dir = \text{From VP} \wedge dir_c = \text{To VP} \\ \text{False} & \text{otherwise} \end{cases} \quad (8)$$

3 Results

This section presents the achieved results and methods of evaluation of the algorithms, which were presented above. The speed of video processing is also discussed.

The presented traffic analysis system was evaluated on several video streams. The processed video streams have resolution 854×480 and the video camera was located several meters above the road. The angle of the video camera varies as Figure 8 shows.



(a) ROC curve

(b) Precision-Recall curve

Figure 7: ROC and Precision-Recall curves for detection and tracking of vehicles in video. Configuration providing the best results has F-Measure equal to 0.915 and is marked by red color.

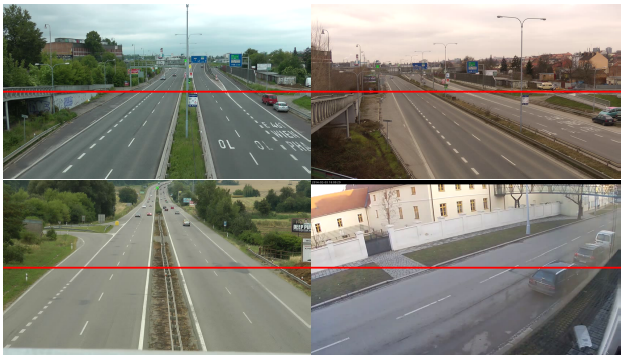


Figure 8: Examples of videos for detection and tracking evaluation. Virtual line which was used for manual ground truth annotation is drawn by red color.

3.1 Detection and Tracking

A manually annotated dataset was created for the evaluation of accuracy of the detection and tracking of vehicles. Imaginary line, see Figure 8, which is crossing the center of image and dividing frames into two equal parts was displayed and for each car, the location and time of crossing the line was annotated. Almost 30 minutes of video was annotated in this way resulting in almost 800 vehicles in the dataset.

The comparison with the ground truth annotation was performed in the following way. For each vehicle which was detected by the traffic analysis system, the trajectory is approximated by a line and the intersection of the approximation with the imaginary line is computed. A match with the ground truth is a vehicle which has trajectory with close intersection to the ground truth intersection and projected time of passing this intersection does not differ too much. If there are more vehicles which satisfy this condition, the vehicle with the smallest time and intersection difference is selected as the match with the ground truth. This way of evaluation was selected because the system targets mainly on overall statistics of passed vehicles.

Nine various configurations which have different maximal distance to the first vanishing point and minimal passed distance of a vehicle were created and evaluated. The ROC and Precision-Recall curves are in Figure 7. Configuration providing the best results has F-Measure [19] equal to 0.915 (Precision is 0.905 and Recall 0.925). The False Negative cases are caused mainly by vehicle occlusions. The occlusions are caused either by a shadow which connects vehicles into one connected component or by a situation when a vehicle partially covers some other vehicle. The False Positives are caused primarily by the motion detection incorrectly dividing a vehicle into two objects and both these objects are tracked and treated as vehicles.

3.2 Direction Estimation and Lane Detection

Several video sequences with a sufficient number of cars were obtained and stability of detected lanes was evaluated for these videos. The results of the evaluation are in Figure 9 and as the graphs show, the detected lanes are almost totally stable and do not change with passing cars. It should be noted that the detected lanes are recomputed always after next 200 cars were observed. Also the directions of the lanes were correctly detected as shown in Figures 9 and 10.

3.3 Evaluation of Speed

Processing speed of the system was also evaluated and the results are in Table 1. The measured framerates include also reading and decoding the video. The system was evaluated on a machine with Intel Dual-Core i5 1.8 GHz and

resolution	traffic intensity	FPS
854×480	high	57.97
	low	82.43
1920×1080	high	28.59
	low	47.88

Table 1: Processing speed evaluation. Approximately 110 minutes of video were used for the evaluation. The videos were divided into groups with respect to the traffic intensity. It should be also noted that the system uses video stream downsampled to ~ 10 FPS, so that the movement is detectable and measurable.

8GB DDR3 RAM. As the table shows, the system can be used for real-time analysis of Full-HD traffic surveillance video. The framerates are higher in videos with lower traffic intensity. The video sequences with higher traffic intensity contain more motion and vehicles which need to be tracked; therefore, more computational resources are used.

4 Conclusions

This paper presents a system for fully automated traffic analysis from a single uncalibrated camera. The camera is automatically calibrated, vehicles are detected, tracked and their direction is computed. Also, the lanes are detected and therefore cars travelling in the wrong way can be detected. The system works in real time and in a fully automated way and therefore it can be used for online traffic analysis with any camera which is monitoring a highway or a street. The system is ready for deployment and it is currently used for online traffic analysis.

The system is able to work under bad lightning and weather conditions. However, for example at night or during rainy weather, the accuracy of detection and tracking decreases slightly because of light reflections from the road. On the other hand, the initialization process can be performed at night without any problem, it will just take longer time because there is a lower amount of vehicles on streets at night.

The main contribution and advantage of the proposed traffic analysis system is that the system works without any manual input whatsoever and the set of conditions which a trajectory of a moving object in video is considered to be a vehicle. Future development of the system will focus mainly on complex crossroads and shadow elimination. Also, elimination of pedestrians from statistics should be addressed.

References

[1] C. Aydos, B. Hengst, and W. Uther. Kalman filter process models for urban vehicle tracking. In *Intelligent Transportation Systems, 2009. ITSC '09. 12th International IEEE Conference on*, pages 1–8, Oct 2009.

[2] D. Beymer, P. McLauchlan, B. Coifman, and J. Malik. A real-time computer vision system for measuring traffic parameters. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 495–501, 1997.

[3] F.W. Cathey and D.J. Dailey. A novel technique to dynamically measure vehicle speed using uncalibrated roadway cameras. In *Intelligent Vehicles Symposium*, pages 777–782, 2005.

[4] Hsu-Yung Cheng and Shih-Han Hsu. Intelligent highway traffic surveillance with self-diagnosis abilities. *Intelligent Transportation Systems, IEEE Transactions on*, 12(4):1462–1472, Dec 2011.

[5] Benjamin Coifman, David Beymer, Philip McLauchlan, and Jitendra Malik. A real-time computer vision system for vehicle tracking and traffic surveillance. *Transportation Research Part C: Emerging Technologies*, 6(4):271 – 288, 1998.

[6] Rong Dong, Bo Li, and Qi-mei Chen. An automatic calibration method for PTZ camera in expressway monitoring system. In *World Congress on Computer Science and Information Engineering*, pages 636–640, 2009.

[7] Yuren Du and Feng Yuan. Real-time vehicle tracking by kalman filtering and gabor decomposition. In *Information Science and Engineering (ICISE), 2009 1st International Conference on*, pages 1386–1390, Dec 2009.

[8] M. Dubská, A. Herout, R. Juránek, and J. Sochor. Fully automatic roadside camera calibration for traffic surveillance. Submitted to: *IEEE Transactions on ITS*, 2014.

[9] Markéta Dubská and Adam Herout. Real projective plane mapping for detection of orthogonal vanishing points. In *British Machine Vision Conference, BMVC*, 2013.

[10] George S. K. Fung, Nelson H. C. Yung, and Grantham K. H. Pang. Camera calibration from road lane markings. *Optical Engineering*, 42(10):2967–2977, 2003.

[11] T. Horprasert, D. Harwood, and L. S. Davis. A statistical approach for real-time robust background subtraction and shadow detection. In *Proc. IEEE ICCV*, volume 99, pages 1–19, 1999.

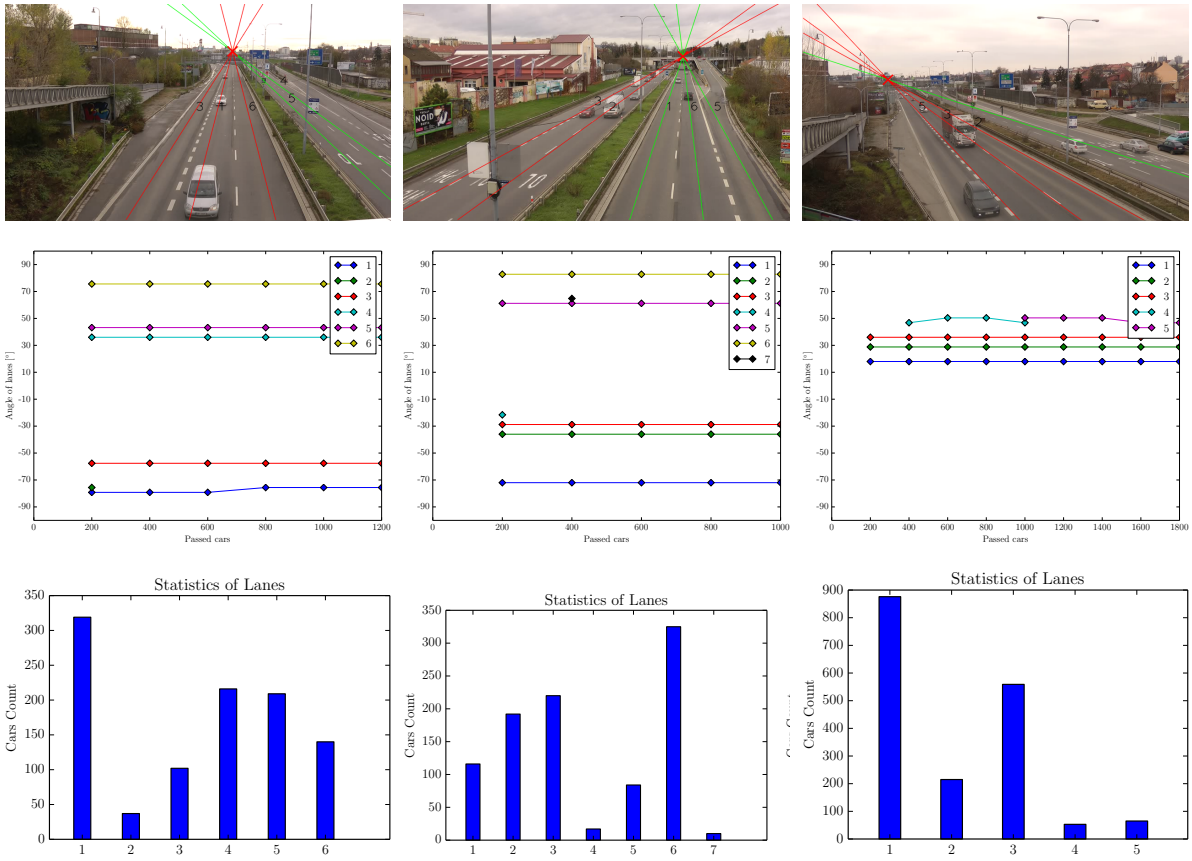


Figure 9: Stability of lanes detection for long video sequences. The top line of images presents the detected lanes. Only lanes which were valid for the last frame of video are drawn. The middle images show changes in detected lanes over time as new cars were observed in video. Finally, the bottom line shows the statistics of observed cars in the detected lanes.

[12] Jun-Wei Hsieh, Shih-Hao Yu, Yung-Sheng Chen, and Wen-Fong Hu. Automatic traffic surveillance system for vehicle tracking and classification. *Intelligent Transportation Systems, IEEE Transactions on*, 7(2):175–187, 2006.

[13] Lili Huang. Real-time multi-vehicle detection and sub-feature based tracking for traffic surveillance systems. In *Informatics in Control, Automation and Robotics (CAR), 2010 2nd International Asia Conference on*, volume 2, pages 324–328, March 2010.

[14] Young-Kee Jung and Yo-Sung Ho. Traffic parameter extraction using video-based vehicle tracking. In *Intelligent Transportation Systems, 1999. Proceedings. 1999 IEEE/IEEJ/JSAI International Conference on*, pages 764–769, 1999.

[15] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME – Journal of Basic Engineering*, (82 (Series D)):35–45, 1960.

[16] Neeraj K Kanhere, Stanley T Birchfield, and Wayne A Sarasua. Automatic camera calibration using pattern detection for vision-based speed sensing. *Journal of the Transportation Research Board*, 2086(1):30–39, 2008.

[17] Dieter Koller, Joseph Weber, and Jitendra Malik. Robust multiple car tracking with occlusion reasoning. pages 189–196. Springer-Verlag, 1993.

[18] A. H S Lai and N. H C Yung. Lane detection by orientation and length discrimination. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 30(4):539–548, Aug 2000.

[19] Christopher D MANNING, Prabhakar RAGHAVAN, and Hinrich SCHÜTZE. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[20] J. Melo, A. Naftel, A. Bernardino, and J. Santos-Victor. Detection and classification of highway lanes using vehicle motion trajectories. *Intelligent Transportation Systems, IEEE Transactions on*, 7(2):188–200, June 2006.

[21] B. Morris and M. Trivedi. Robust classification and tracking of vehicles in traffic video streams. In *In-*

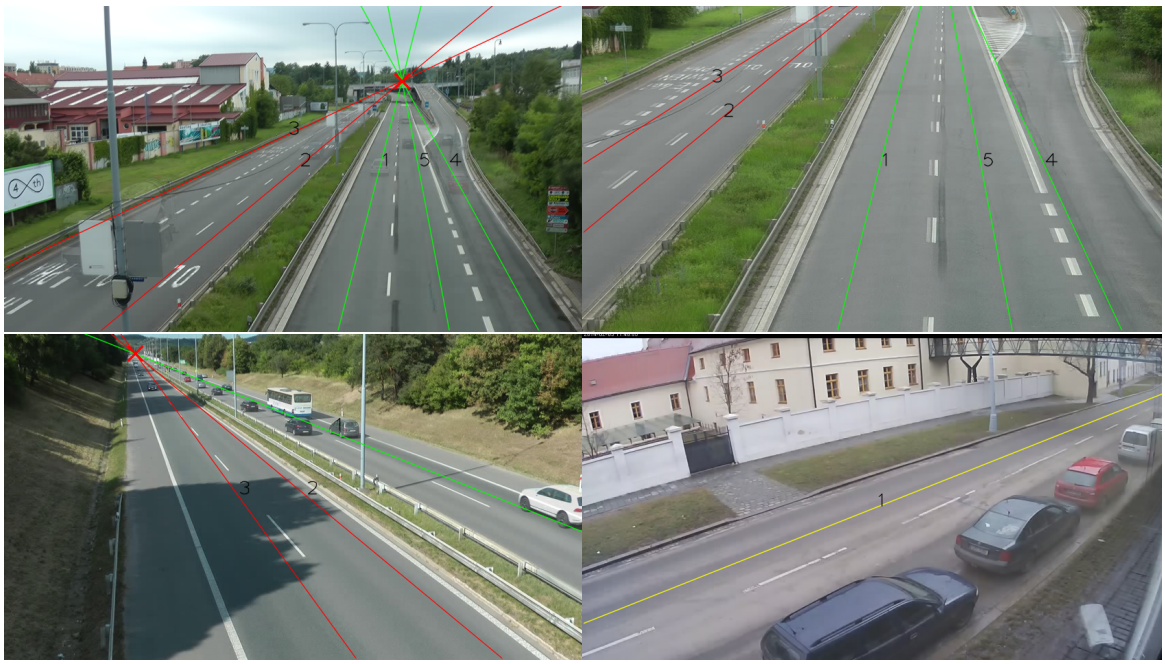


Figure 10: Example of detected lanes and dominant direction of cars in the lanes. Green color means that the majority of cars is heading towards the first vanishing point and red the opposite. Yellow color means that there is no dominant direction for the given lane. Example of this situation is shown in bottom right image. It should be noted, that the centers of cars, which are used for lanes detection, are not in the middle of the lanes because of the angle of view.

telligent Transportation Systems Conference, 2006. ITSC '06. IEEE, pages 1078–1083, 2006.

[22] Roya Rad and Mansour Jamzad. Real time classification and tracking of multiple vehicles in highways. *Pattern Recognition Letters*, 26(10):1597 – 1607, 2005.

[23] T.N. Schoepflin and D.J. Dailey. Dynamic camera calibration of roadside traffic management cameras for vehicle speed estimation. *IEEE Transactions on Intelligent Transportation Systems*, 4(2):90–98, 2003.

[24] J. Shi and C. Tomasi. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*, pages 593–600, Jun 1994.

[25] Kai-Tai Song and Jen-Chao Tai. Dynamic calibration of Pan–Tilt–Zoom cameras for traffic monitoring. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 36(5):1091–1103, 2006.

[26] C. Stauffer and W. E. L. Grimson. Adaptive background mixture models for real-time tracking. In *Computer Vision and Pattern Recognition*, volume 2, pages 246–252, 1999.

[27] Carlo Tomasi and Takeo Kanade. *Detection and tracking of point features*. School of Computer Science, CMU, 1991.

[28] B.L. Tseng, Ching-Yung Lin, and J.R. Smith. Real-time video surveillance for traffic monitoring using virtual line analysis. In *Multimedia and Expo, 2002. ICME '02. Proceedings. 2002 IEEE International Conference on*, volume 2, pages 541–544 vol.2, 2002.

[29] Kunfeng Wang, Hua Huang, Yuantao Li, and Fei-Yue Wang. Research on lane-marking line based camera calibration. In *International Conference on Vehicular Electronics and Safety, ICVES*, 2007.

[30] Zhaoxiang Zhang, Tieniu Tan, Kaiqi Huang, and Yunhong Wang. Practical camera calibration from moving objects for traffic scene surveillance. *IEEE Transactions on Circuits and Systems for Video Technology*, 23(3):518–533, 2013.

[31] Z. Zivkovic. Improved adaptive gaussian mixture model for background subtraction. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 2, pages 28–31 Vol.2, 2004.

Custom Unmanned Aerial Vehicle for Photography based Terrain Reconstruction

Jernej Kranjec*
Supervised by: Borut Žalik†

University of Maribor
Faculty of Electrical Engineering and Computer Science
Laboratory for Geometric Modelling and Multimedia Algorithms
Smetanova ulica 17, SI-2000 Maribor / Slovenia

Abstract

Consumer electronics have become considerably powerful in terms of hardware features, which can be expanded beyond their original design with custom software.

This paper serves as a summary of a student's experience of acquiring suitable aerial images for terrain reconstruction. The paper covers the utilization of consumer components like a cell phone with on-board sensors, point-and-shoot cameras and a prefabricated model airplane, combined with easily accessible electronics. This was to create an inexpensive platform for high definition aerial photography, as needed for terrain reconstruction. It describes the challenges of building such a platform and presents an overview of the results.

Keywords: unmanned aerial vehicle, aerial photography, terrain reconstruction

1 Introduction

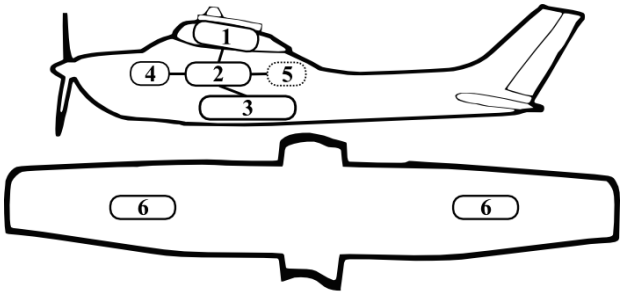
Today consumer devices are more powerful and flexible than ever. By considering the increased popularity of easily accessible hobby-grade remote-operated models, we decided to combine these within a customized unmanned aerial vehicle, adapted for carrying a customized stereo camera rig for aerial photography. Our focus on consumer devices was driven primarily by their prices and availability. By utilizing our skills we turned them into a platform capable of performing features normally found within professional kits.

The current iteration is based on a scaled foam model of a Cessna airplane, which offers a lot of space for necessary modifications and payload. The proposed control unit is split into an Android phone, which is used for sensors, computing and communications, and a control module from custom electronics for controlling the servo-motors used in the model. Two Canon point-and-shoot budget cameras were used for stereo camera mounting. The cameras were chosen due to their support of customized third-party firmware.

This enabled the acquiring of high-resolution aerial images of a desired area with high overlap, which were then used for terrain reconstruction.

2 Model setup

A scaled-down model of a Cessna 182 airplane was chosen made of durable Elapor foam, with a wingspan of 1400mm. This provided a lightweight durable base that could be easily modified, with enough room for additional gear. The model was equipped with a KORA 10-15 brushless electrical engine with a HobbyWing 40A speed regulator driving an APC 11x5.5in propeller, powered by a 2200mAh LiPo battery. The model was controlled by a FrSky 8 channel X8R Remote Control receiver with telemetry feedback, which operates in the 2.4GHz frequency band, allowing over 1km line of sight operational range. 6 standard 9g servo motors were used to operate the control surfaces of the model. A structural diagram of the modified model can be seen in Figure 1.



1. cell phone with panorama attachment
2. control module
3. battery
4. RC-receiver
5. servo-motors
6. point-and-shoot cameras

Figure 1: Aircraft model structural diagram

The model was set up in the following configuration: channel 1 servo operates the vertical stabilizer, channel 2

*jernej.kranjec@gmail.com
†borut.zalik@um.si

servo operates the horizontal stabilizer, channel 3 controls the engine output, channels 4 and 5 servos operates the ailerons, channels 6 and 7 servos supervise the flaps and channel 8 the spare servo-channel used to signal who is in control.

Modifications performed on the model include mounting the stereo camera rig (see Figure 3) through the fuselage to preserve the model's center of gravity, securing it onto the wings with custom 3D printed holders (see Figure 2) in order to prevent the camera rig from being damaged during takeoffs and landings.

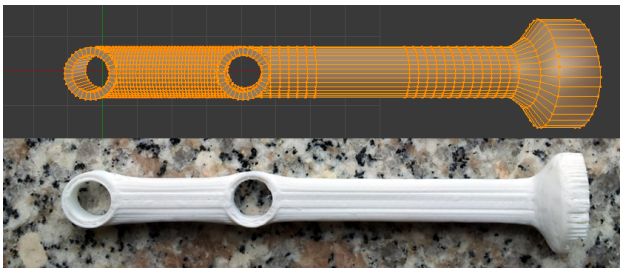


Figure 2: Customized holder, designed in Blender and printed on RepRap 3D printer

Figure 3: Camera rig mounted onto the wings

The battery bay was extended to accommodate a bigger battery pack and fine tune the model's center of gravity by shifting its position as necessary. Stock landing gear was replaced with one made out of carbon fiber mounted on flexible FR-4 fiber glass strip (see Figure 4), in order to better absorb and withstand the increased weight of the model.

Figure 4: Carbon fiber landing gear with extended battery bay

Lastly, attaching a customized cell phone case and holder carved into the wing on the top of the fuselage with a modified Kogeto DOT 360° panoramic lens attachment, for capturing video of the model in flight for later review and visualization.

The complete setup (see Figure 5), including the cameras weights 1.9kg, provides 5 minutes of flying time, out of which 2-3 minutes are at the minimum desired altitude difference of 200m for taking aerial photographs. The total traveled distance of the model using that configuration is about 4.5km at an average ground speed of around 50km/h.

Figure 5: Photo of the model ready for take-off

3 Electronics

First part of the control unit was based around the Samsung Galaxy S2 cell phone. The device was chosen because of the available on-board sensors containing a triple-axis accelerometer, triple-axis gyroscope and a triple-axis compass, which were used as an Inertial Measurement Unit in the autopilot implementation in order to determine the relative orientation of the model. A GPS receiver for locational logging and navigation and a GPRS modem for communication with the ground computer and real-time visualization of the flight. It also contained an 8MP camera with video capabilities used for recording in-flight video of the model, as well as providing a 1GHz dual core computer with 1GB of RAM within a programmable-friendly environment.

For the second part a control module was constructed from custom electronics, which took the control inputs from either the phone or the RC-receiver and performed the actual control of the servos on the model. This allowed for controlling of the servos from the phone as well as the remote control.

The customized electronics consisted of an Atmega128 microcontroller from Atmel, an FT230x USB-serial bridge from FTDI, and some passive components. The Atmega128 was chosen because of its hardware support for driving a large number of servo-motors using its timer modules. It also provided pin change interrupts, which

were used to read the outputs from the RC-receiver. The signals generated and read by the microcontroller are standard servo control signals with a 50Hz period and a duty cycle of 1-2ms, where 1ms represents -45 servo rotation from center and 2ms corresponds to 45 rotation. The On-The-Go USB capabilities of the phone were used to connect the control module to the phone, which allows connection of the device to the phone. We utilized the serial interface of the microcontroller to connect the phone to the control module using a USB-serial bridge. We used the serial connection simplified programming of the module as it negated the need for any high-level abstraction requested by the USB protocol. The electronics block diagram can be seen in Figure 6.

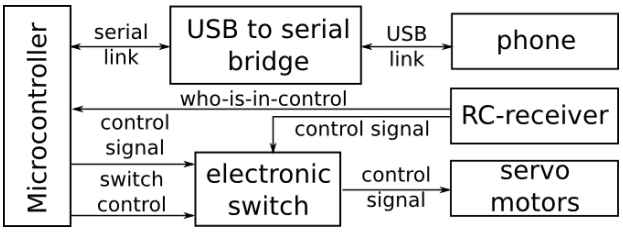


Figure 6: Control module block diagram

4 Camera setup

Two Canon A2200 cameras were chosen for this setup as they were inexpensive, light, and allowed us to extend their functionalities using third party firmware. Our first step was porting the open-source firmware CHDK on to the camera. This allowed the usage of the camera's parameters and settings, which are usually hidden within the consumer firmware, like manual focus and exposure control. We also acquired new features, among others the ability to save raw 12bit images, run custom scripts, and synchronize the shutter release across multiple cameras.

The following method is employed to test the camera's ability to capture synchronized stereo images. A rotating platform was used, constructed of a small electric motor and a CD-ROM holder, which held a CD-ROM, onto which an LED with a battery was glued and balanced. We took the first picture with a known exposure time of 20ms. Using the light trail left on the image, it was possible to calculate the rotational speed of the setup, which was 10.36Hz. Knowing this, pictures of the rotating platform with both cameras shutters synchronized were taken and the light trails on both pictures took into account the start and end-points, and its length was then compared. Using multiple measurements the deduction was made that the synchronization between cameras were less than 0.1ms apart. Assuming a speed of 50km/h for the model, the cameras would trigger in less than 1.4mm of travel between each other, making the perspective distortion minimal. Testing the cameras can be seen in Figure 7.

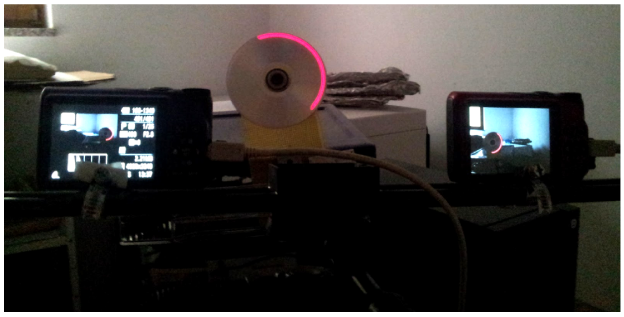


Figure 7: Camera synchronization testing rig

In order to construct the stereo camera rig, 1m long prefabricated 1cm diameter hollow carbon fiber tubes were used. The carbon fiber tubes were perfectly aligned parallel to each other, placed 40mm apart and glued onto a supporting plastic platform, which operated as a camera stand mount. In order to attach and align the cameras, a cradle (see Figure 8) was made out of larger tubes, which held a standard 3/4in camera mount screw.

Figure 8: Camera cradle for the stereo rig

5 Software

Our autopilot with ground communications and sensor logging was implemented as an Android application. This allowed usage of the underlying Android APIs, making the development easier.

Using Android's Motion Sensors API, which performed sensor fusion on the on-board sensors, the orientation of the device was obtained according to the device's coordinate system. By strategically placing the phone on the model, the relative orientation and direction of the model was made in the form of angles around the device's axis.

The GPS data, location, altitude, and speed of the device were collated using the Android's Location API. The API also allowed navigation tasks to be performed like calculating the distance or heading to a specified point, which the autopilot had to reach.

The implementation of the autopilot was designed around a Proportional Integral Derivative feedback loop or a PID controller, which takes the device's current orientation and desired orientation as input, and provides servo rotation as output. A separate PID controller for each of the models control axes was used, thus translating the device's coordinate system to yaw, pitch and roll as well as throttle.

The autopilot was developed and tested using a FlightGear flight simulator, which can simulate the sensors found in the phone. In order to simulate the flight, FlightGear's two-way network capabilities for sensor input and servo output was used, as required and produced by the autopilot application. Simulated flights were conducted using the FlightGear's Rascal 110 RC model. This provided a crash-proof environment for testing and tuning. A testing session with live visualization over the network from the autopilot application can be seen in Figure 9.

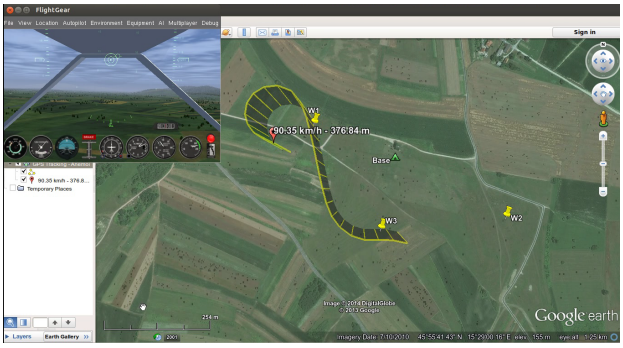


Figure 9: Autopilot testing with live visualization

The visualization was done using custom software made in Python. When using it, the logged GPS data from the autopilot is aggregated. The output is a Google Earth compatible Keyhole Markup Language structured file, containing a flight path, ground speed, altitude, GPS resolution and way points used by the autopilot for navigation. All of the sensor data was rendered into separate videos, which were later combined and synchronized with the panoramic video from the phone's camera and visualization playback from Google Earth (see Figure 10).

Figure 10: Visualization of flight data

Ground communication was done through the Internet using the built-in GPRS modem of the phone. GPS data from the autopilot was processed by the software and piped in KML format into Google Earth where it was rendered in real time.

Unrolling of the 360° panoramic images and video taken with the Kogeto DOT attachment was done using Log-Polar to Cartesian conversion with an added scaling factor to offset the distortion of the lens. The final image was further improved with B-spline interpolation [3]. The final results can be seen in Figure 11.

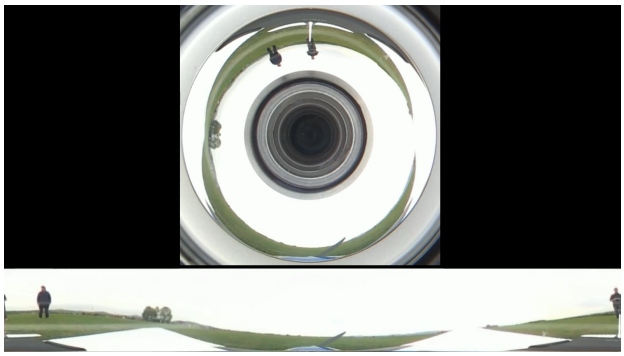


Figure 11: Raw and unrolled 360° panoramic image

6 Terrain reconstruction

From the camera rig we obtained 14MP stereo images, taken 5 seconds apart. In order to ensure adequate overlap between images, only those taken at a minimum altitude difference of 200m or more were used. The selected images were then processed to remove lens distortion detected by taking photos of calibration checkered board with each of the used cameras.

To test whether the aerial images are suitable for terrain reconstruction, the following tool chain was employed. First we used VisualSFM [1, 6] which implemented SiftGPU [5] for feature matching between images, calculated the camera positions within the world coordinate system and performed the sparse terrain reconstruction using the Structure from Motion method. That information was then used for two more applications; CMPMVS [2] and SURE [4]. In order to perform dense reconstruction, the CMPMVS application uses the augmented Labatut CGF 2009 method, while the SURE application applied an improved Semi-Global Matching (SGM) algorithm.

7 Results

A test flight covering around 170,000m^2 was performed, which generated 38 useful images (see Figure 12). After processing the images the result were obtained for the area seen in Figure 13.

Figure 12: Pictures used in reconstruction

Figure 13: Ortho-photo of the reconstructed terrain

VisualSFM produced a point cloud consisting of 1 million points, resulting in a resolution of approximately 5.9 points per square meter. A section of the generated point cloud can be seen in Figure 14.

CMPMVS produced a point cloud of 2.3 million points, resulting in a resolution of approximately 13.7 points per square meter. A section of the generated point cloud can be seen in Figure 15.

SURE produced a point cloud for every image pair, which when combined form a point cloud with 43.7 million points, resulting in a resolution of approximately 257.3 points per square meter. A section of the generated point cloud can be seen in Figure 16.

8 Conclusions

The constructed unmanned aerial vehicle platform performed well enough to successfully perform initial testing, but there are of course some drawbacks. Namely, this configuration of the model requires a decent landing strip for takeoffs and landings. Battery life is an issue due to weight constraints. The current version of the autopilot would be good enough for simple flyovers and return-to-home, but not stable enough for precise maneuvers to acquire clear images.

Future work on this platform will concentrate on upgrading the used airplane with a bigger model wthin a pusher configuration where the propeller is mounted behind the engine, and the engine itself will be mounted on top of the fuselage. This will remove the need for a landing gear and landing strip, since it will be possible to launch the model by hand and land it anywhere without damage. Also an update of the customized electronics with a pass-through from receiver to the phone would allow a fly-by-wire type autopilot, whereby the autopilot would adjust the control surfaces of the airplane to maintain the position input by the ground controller.

References

[1] Brian Curless Changchang Wu, Sameer Agarwal and Steven M. Seitz. Multicore bundle adjustment. In *CVPR*, pages 3057–3064. IEEE, 2011.

[2] Michal Jancosek and Tomas Pajdla. Multi-view reconstruction preserving weakly-supported surfaces. In *CVPR*, pages 3121–3128. IEEE, 2011.

[3] Jernej Kranjec and Božidar Potočnik. Razvijanje 360° panoramske slike iz leče s sferičnim zrcalom (in slovene - english title: Unrolling a 360° panoramic image from a spherical mirror lens). In *ROSUS*, pages 129–136. University of Maribor - Faculty of Electrical Engineering and Computer Science, 2013.

[4] Dieter Fritsch Mathias Rothermel, Konrad Wenzel and Norbert Haala. Sure: Photogrammetric surface reconstruction from imagery. In *Proceedings LC3D Workshop, Berlin*, 2012.

[5] Changchang Wu. Siftgpu: A gpu implementation of scale invaraint feature transform (sift), 2007.

[6] Changchang Wu. Visualsfm: A visual structure from motion system, 2011.

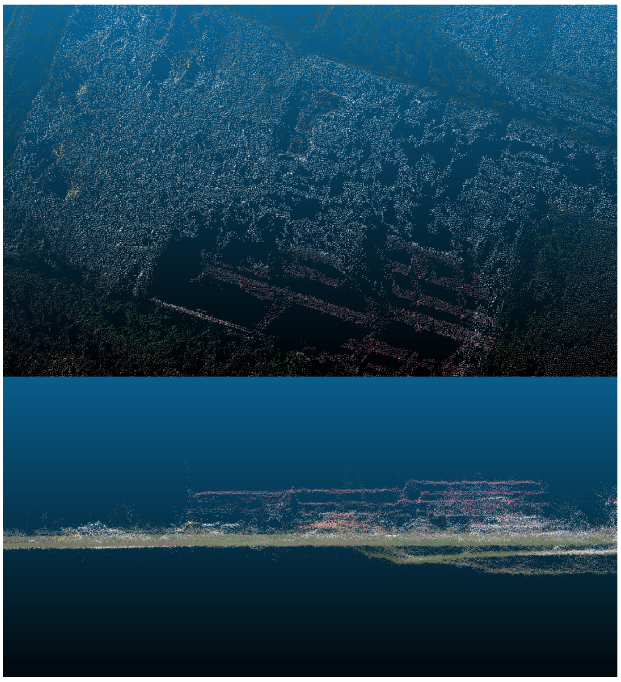


Figure 14: Point cloud generated by VisualSFM

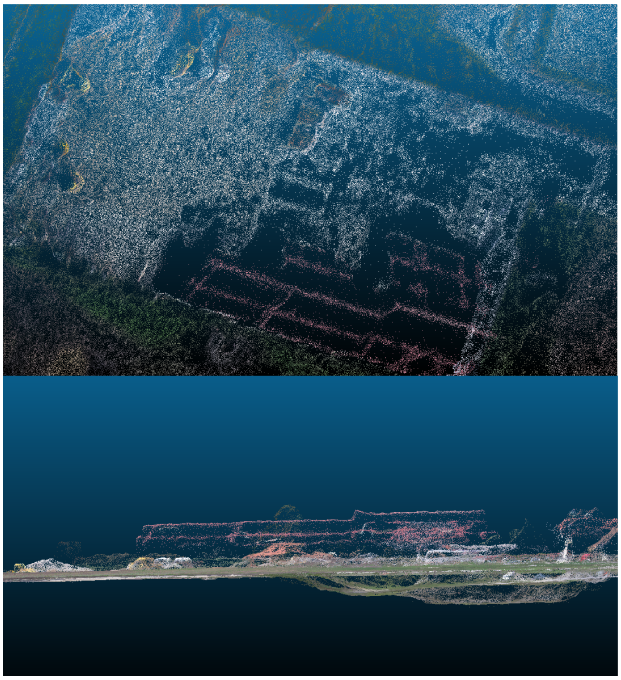


Figure 15: Point cloud generated by CMPMVS

Figure 16: Point cloud generated by SURE

Coastal Monitoring for Change Detection Using Multi-temporal LiDAR Data

Denis Kolednik*
Supervised by: Domen Mongus†

University of Maribor Faculty of Electrical Engineering and Computer Science
Laboratory for Geometric Modelling and Multimedia Algorithms
Smetanova ulica 17, SI-2000 Maribor / Slovenia

Abstract

Monitoring changes in landscapes is important for several environmental and geographical studies. This paper considers a coastline change detection approach using multi-temporal data captured by Light Detection And Ranging (LiDAR) technology. The proposed method consists of four steps. In the first step a heightmap is generated from LiDAR data. The second step represents the core of the proposed method. In this step dense optical flow of the multi-temporal heightmaps is computed, yielding motion vectors for each point. In the third step, points with similar motion vectors are clustered. In the last step a displacement for each cluster is estimated, representing the movement of soil. An evaluation of the approach shows a 91.897% accuracy when estimating displacements and a 93.710% accuracy when detecting displaced areas.

Keywords: change detection, optical flow, LiDAR

1 Introduction

Monitoring coastal changes is an important task for several studies. It is interesting from a geographical perspective to study the trends of local landscape changes. Such studies are also important for the economy. Various area utilisations can be efficiently planned by having priori knowledge about certain landscapes and their changing tendencies. Coastal areas represent a very dynamic case regarding landscapes. Constant tidal activity washes up and washes away soil from the surface. Such behaviour results in an ever-changing shape of coastline, especially in respect to salt pans. Frequent evaporation and flooding of such areas cause an accelerated process of the previously-mentioned surface changes. Such areas have many changes in the short term and are, therefore, excellent test sets for the change detection approaches of coastal data.

This paper presents an approach for the detection and estimation of coastal surface changes over a certain timespan. Surface data were obtained by LiDAR technology. The result of the presented approach are displacement estimations of certain parts of the coastline, where each point belonging to the displaced area is labelled. The paper is organised into 6 sections. Section 2 gives a short overview of related work. Section 3 provides a short summary on data acquisition and the test area. Section 4 describes the procedure for estimating surface change detection. Section 5 presents the results of testing. A summary of the paper is given in Section 6.

2 Related work

Many studies are engaged in coastal monitoring. Most of them focus on extracting the coastline and not computing the actual changes. Xu-kai, Xia, Qiong-qiong and Ali Baig [20] proposed an approach for automated coastline extraction using the Otsu algorithm and Canny edge detection [5]. Bouchama and Yan [4] proposed an approach for detecting changes between two datasets using a window-to-window comparison and SURF features for alignment. Bo, Dellepiane and De Laurentiis [3] extracted the coastline using an approach based on the local contextual information of remotely sensed data. Niedermeier, Romaneen and Lehner [16] detected the coastline using an approach based on wavelet methods. Alesheikh, Ghorbanali and Nouri [1] present an approach for coastal change detection based on a combination of histogram thresholding and band ratio techniques. A semi-automatic approach based on fuzzy connectivity concepts for the coastline extraction from SAR images was proposed by Dellepiane, De Laurentiis and Giordano [7]. Ali [2] proposed an approach that represents coastlines as curves that are divided into segments of the same length in a multi-temporal dataset. At each such corresponding segment, between the preliminary and postliminary acquired data, an Euclidean distance is computed that represents the amount of coastal displacement.

*denis.koeldnik@um.si
†domen.mongus@um.si

3 Data acquisition

The data used in this paper were obtained by the laser remote sensing technology called airborne LiDAR. It is considered to be the most advanced remote sensing technology at the moment. Conceptionally it is similar to a RADAR or SONAR, with the main difference being the wavelength of the signal. LiDAR emits a series of laser pulses towards the surface where they reflect and travel back to the device. As the speed of light is constant, the distance can be determined by the time it takes the pulse to travel from and back to the device. Such a procedure is repeated under different scan angles so that it forms a line, as seen in Figure 1. Having a capture frequency of over 200.000 pulses per second and density of more than 40 points per square metre [18], a very detailed representation of the world can be obtained. LiDAR technology is also capable of penetrating through vegetation and recording the terrain beneath. The data are georeferenced using a GPS system for positioning and an inertial measurement unit (IMU) for angle determination of the emitted pulses [14, 12]. Data are stored as a three-dimensional point-cloud without a topology. Water areas, on the other hand, are troublesome, as low reflectance of light on water results in a low number of acquired points [19].

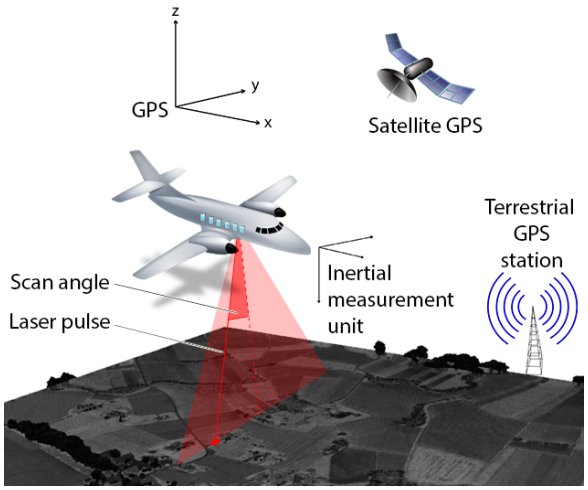


Figure 1: Data acquisition with LiDAR technology.

The data for testing the proposed approach were obtained at the Seovlje salt pans located near Portoro, Slovenia. Data were obtained in the years 2008 and 2010 and is already classified.

4 Surface change detection

Surface movement in geography is considered as a visible sliding of soil from its original position [21]. Changes on the coast can be considered as movement of the surface towards or away from the current coastline. An example is shown in Figure 2. In such a case, a rigid translation can be considered. The proposed approach is based on this assumption. A rigid translation is considered as a gradual changing of the coastline's shape. This assumption does not suffice for coasts that have suddenly changed the curvature of the coastline such as man-made embankments or excavations of an area. The approach consists of four main steps, as described in the next four subsections.

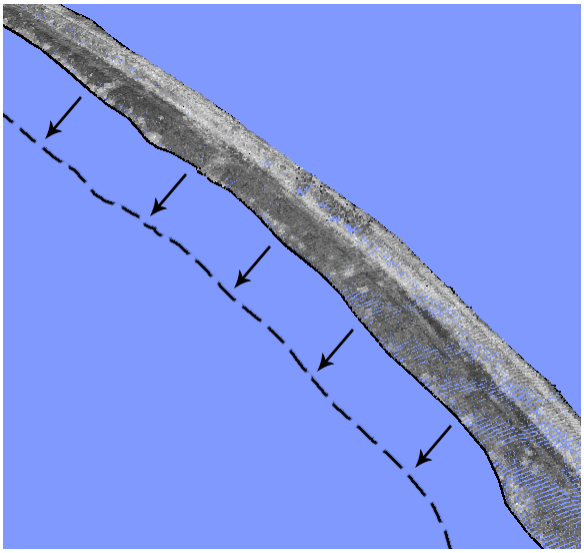


Figure 2: Displacement of the coastline.

4.1 Heightmap

In the first step a heightmap is generated from the input point-cloud. The resolution of the heightmap is dependent on the density of the obtained LiDAR data. The mapping requires an interpolation method, as points from the point-cloud do not coincide with the points on the heightmap. The inverse distance weighting (IDW) method [6] is used in the proposed approach. Using this interpolation method a height value is estimated using information from neighbouring points. The weight of each point within the neighbourhood is inversely-proportional to the power of the distance. A greater number of neighbours results in a smoother heightmap. The neighbourhood of the nearest 5 points is used for the proposed approach. The result of a heightmap is shown in Figure 3.

4.2 Optical flow

The calculation of optical flow is the key step of the proposed approach. It represents a motion estimation of objects within a scene and is defined as an apparent motion of intensity patterns on the scene [17]. Sparse and dense optical flows exist. A well known method for sparse optical flow was proposed by Lucas and Kanade [11]. It estimates motion vectors at feature points. Dense optical flow, on the other hand, provides a motion field consisting of motion vectors for each heightmap point. Such methods were

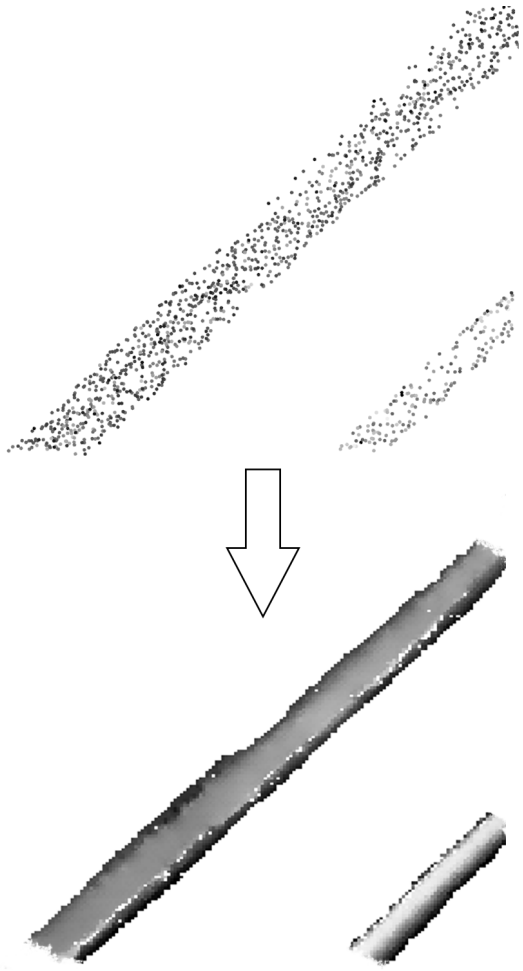


Figure 3: The representation of point cloud mapping to a height grid. The lower image represents a heightmap of the upper point cloud.

presented by Horn and Schmuck [9] and by Farneb [8]. The latter is used in the proposed approach as it yields more robust results.

The main idea of the optical flow computation procedure proposed by Farneb is to find a best fit between the point neighbourhoods of two heightmaps. Each neighbourhood is represented by a quadratic polynomial. Such a polynomial is obtained from polynomial expansion [8], based on normalised convolution, as proposed by Knutsson and Westin [10]. By assuming that the motion is gradual, a certain area of the corresponding point in the postliminary acquired heightmap is examined for the best fitting polynomial. In the case of the motion being sudden, meaning that is does not progress slowly over time, may lead to less accurate motion estimation. The distance between the polynomial on the first heightmap and the best fitted polynomial on the second represents the size of the motion vector. The angle of the vector is computed using the following equation:

$$\theta = \arctan\left(\frac{p_{2y} - p_{1y}}{p_{2x} - p_{1x}}\right). \quad (1)$$

Figure 4: A motion vector field.

A field of such vectors represent those motions that have occurred between two heightmaps over a certain time span, as shown in Figure 4. At this point the motion vectors are mapped from the heightmap to the point-cloud.

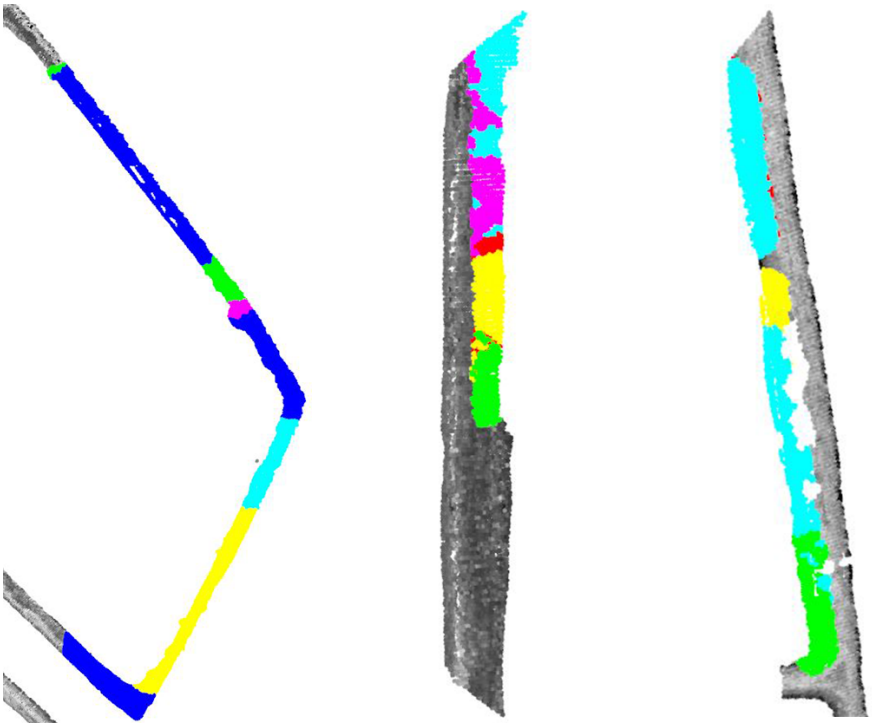
4.3 Point clustering

The result of the procedure described in 4.2 is a field of vectors, each representing motion at a point in the point-cloud. A clustering procedure is proposed for uniting points with similar motion vectors. The proposed clustering uses two criteria for determining whether a point is a part of the cluster or not. The first is the angle of the motion vector. Points with similar motion vector angles are clustered together. As noise within the data distorts motion vector estimation, an angle threshold of 15° is taken into account. The second criteria is the distance between points. A threshold is used for the maximum allowed distance between points. The next step is to find candidate points to cluster. A kd-tree nearest-neighbour search is used to find points within a certain radius [15]. Each newly inspected point that meets the criteria is added to the cluster and triggers a recursive method for finding new nearest-points. The procedure is finished when all the points have been inspected. In order to prevent false detections caused by noise, a threshold is introduced for a minimal number of points in a cluster. An example of clustering on test data is shown in Figure 5.

The goal of the proposed approach in this paper is to find changes of coastal surface. The computed clusters represent parts of land that had moved from their original positions.

4.4 Displacement of clusters

After clustering is finished, the positions must be determined as to where the clusters have moved on the postliminary acquired point-cloud. As each point has its own motion vector, an end position can be computed straightforwardly by adding the motion vector to the point coordinate. It is unnecessary for the calculated displaced points to represent the actual state of the second point-cloud. A kd-tree nearest-neighbour search is performed for determining the actual displaced points on the second-point cloud. Points within the same cluster have similar motion



(a) B190201 (b) B190204 (c) B204216

Figure 5: Dataset for testing with marked clusters.

vectors. Based on that fact, a global displacement vector for the cluster can be obtained as an average of all the contained vectors. The resulting vector represents the direction and amount of displacement the coastline has undergone.

5 Results

The proposed approach was implemented in C++ using the Qt 5 framework. Tests were performed as a single threaded process on a desktop workstation using the following hardware: Intel i5-3570K and 16GB of DDR3 RAM. The dataset used for testing were point clouds from binary LAS files. The average size of the test-set point-clouds was 100.000 points, and a grid with 0.5m resolution was used. The testing data represent those parts of the Seovlje salt pans that underwent the most change. Only those points recognised as ground were considered, because buildings and vegetation were not the studied subjects and would have unnecessarily slowed down the computation.

Two evaluation metrics were used for the proposed approach. The first evaluated the accuracy of the estimated displacements, while the second evaluated the accuracy of the correctly detected displaced areas. Reference data of surface movements had to be obtained for the purpose of evaluating displacement estimation. This was done by an expert in the field of geography. The tool used for measuring the reference data was LIDARLiVE [13]. As shown in Figure 6, this allowed a clear estimation of displacements by simultaneously displaying both point-clouds within a cross-section view.

The datasets and detected movements are shown in Figure 5. Test results for displacement estimation accuracy on 3 datasets are shown in Table 1. Verification of the obtained results show an average error of 8.103%. A slightly higher error value in dataset B204216 was the consequence of missing larger parts of data in some areas. The same datasets were used for evaluating the detection accuracy. The results are shown in Table 2. The proposed approach achieved an average of 93.710%.

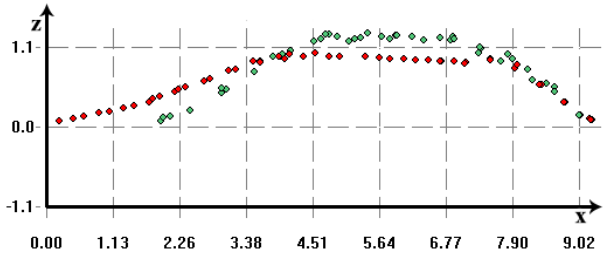


Figure 6: Cross-section view of two point clouds.

The results of the proposed approach were compared to the results of the coastline curvature approach introduced by Ali [2]. The same test-set was used for this purpose, however, the measuring of coastline displacements was limited to the areas of clusters that were detected by the proposed approach, due to consistency of result comparison. The values of the displacements and error are

Table 1: Results and comparison of the testing of displacement computation on 3 datasets with detected multiple displaced clusters.

Dataset	Cluster number	Actual displacement (m)	Proposed approach displacement (m)	Proposed approach error (%)	Coastline curvature displacement (m)	Coastline curvature error (%)
B190201						
	1	5.421	5.391	0.557	5.284	2.536
	2	4.966	5.220	5.125	4.662	6.108
	3	4.282	4.719	10.221	3.403	20.539
	4	5.142	5.550	7.938	5.292	2.922
	5	5.238	4.961	5.286	5.080	3.015
B190204						
	1	2.978	2.766	7.120	3.260	9.460
	2	2.390	2.548	6.605	1.916	19.820
	3	1.807	1.634	9.593	1.798	0.520
	4	1.390	1.584	13.964	1.400	0.720
	5	1.623	1.610	0.810	1.569	3.350
B204216						
	1	1.825	1.639	10.178	1.794	1.672
	2	2.035	2.120	4.185	2.148	5.538
	3	2.196	1.820	17.119	2.048	6.757
	4	1.821	1.552	14.749	1.704	6.405

shown in Table 1 under the coastline curvature approach columns. The average error of 6.383% shows that that the approach, proposed in this paper, is 1.720% less accurate than the coastline curvature approach. The reason for this is in the way of representing a shoreline. The proposed approach takes in account a wider area of the coastline, while the coastal curvature uses only the curve of the coastline which results in the coastal curvature approach being more sensitive to noise than the proposed approach.

The results of the proposed approach would be satisfactory for use within undemanding fields but would be inappropriate for tasks needing maximum precision. The main reason for the resulting error is the optical flow procedure, as it is difficult to find global parameter settings.

Table 2: Results of testing the accuracy of displaced areas.

Dataset	Coastline length (m)	Detected coastline length (m)	Error (%)
B190201	354.999	406.612	14.539
B190204	62.245	61,409	1.340
B204216	127.145	123.341	2.992

6 Conclusions

This paper proposed an approach for change detection of coastal surfaces using multi-temporal LiDAR data. The results show the displacement estimations of coastline areas. The part of the approach that is the main bottleneck is optical flow computation, as it is noise sensitive and highly dependent on the parameter settings. Future work should mainly be focused on optimising this part.

References

[1] A.A. Alesheikh, A. Ghorbanali, and N. Nouri. Coastline change detection using remote sensing. *International Journal of Environmental Science & Technology*, 4(1):61–66, 2007.

[2] T.A. Ali. Analysis of shoreline-changes based on the geometric representation of the shorelines in the GIS database. *International Journal of Geography and Geospatial Information Science*, 1(1):1–16, 2010.

[3] G. Bo, S. Dellepiane, and R. De Laurentiis. Semiautomatic coastline detection in remotely sensed images. In *Geoscience and Remote Sensing Symposium, 2000. IGARSS 2000. Proceedings. IEEE 2000 International*, volume 5 of *IGARSS 2000*, pages 1869–1871. IEEE, 2000.

[4] M. Bouchahma and W. Yan. Long-term coastal changes detection system based on remote sensing and image processing around an island. In *Geoinformatics, 2012 20th International Conference on*, pages 1–5. IEEE, 2012.

[5] J. Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698, 1986.

[6] V. Chaplot, F. Darboux, H. Bourennane, S. Leguis, N. Silvera, and K. Phachomphon. Accuracy of interpolation techniques for the derivation of digital elevation models in relation to landform types and data density. *Geomorphology*, 77(12):126–141, 2006.

[7] S. Dellepiane, R. De Laurentiis, and F. Giordano. Coastline extraction from SAR images and a method for the evaluation of the coastline precision. *Pattern Recognition Letters*, 25(13):1461 – 1470, 2004.

[8] G. Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the 13th Scandinavian Conference on Image Analysis*, SCIA'03, pages 363–370, Berlin Heidelberg, 2003. Springer.

[9] B.K.P. Horn and B.G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1-3):185–203, 1981.

[10] H. Knutsson and C.F. Westin. Normalized and differential convolution: Methods for interpolation and filtering of incomplete and uncertain data. In *Proceedings of Computer Vision and Pattern Recognition*, CVPR'93, pages 515–523, 1993.

[11] B.D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, volume 2 of *IJCAI'81*, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[12] D. Mongus and B. alik. Parameter-free ground filtering of LiDAR data for automatic DTM generation. *ISPRS Journal of Photogrammetry and Remote Sensing*, 67(0):1 – 12, 2012.

[13] D. Mongus, S. Penik, and B. alik. Efficient visualization of LiDAR datasets. In *Society of Photo-Optical Instrumentation Engineers*, volume 7513 of *SPIE 2009*, 2009.

[14] D. Mongus and B. Zalik. Computationally efficient method for the generation of a digital terrain model from airborne LiDAR data using connected operators. *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, 7(1):340–351, 2014.

[15] A.W. Moore. An introductory tutorial on kd-trees. Technical report, Computer Laboratory of the University of Cambridge, 1991.

[16] A. Niedermeier, E. Romaneessen, and S. Lehner. Detection of coastlines in SAR images using wavelet methods. *Geoscience and Remote Sensing, IEEE Transactions on*, 38(5):2270–2281, 2000.

[17] P.N. Pathirana, A.E.K. Lim, J. Carminati, and M. Premarante. Simultaneous estimation of optical flow and object state: A modified approach to optical flow calculation. In *Networking, Sensing and Control, 2007 IEEE International Conference on*, ICNSC 2007, pages 634–638, April 2007.

[18] RIEGL Laser Seasurement Systems. Datasheet vq580 18-09-2013, 2014. Available at: http://www.riegl.com.

[19] W. Xiao. Detecting changes in trees using multi-temporal airbone lidar point clouds. Masters thesis, Faculty of geo-information science and Earth observation of the university of Twente, 2012.

[20] Z. Xu-Kai, Z. Xia, L. Qiong-qiong, and M.H. Ali Baig. Automated detection of coastline using Landsat TM based on water index and edge detection methods. In *Earth Observation and Remote Sensing Applications, 2012 Second International Workshop on*, EORSA 2012, pages 153–156, 2012.

[21] M. Zorn and B. Komac. *Zemeljski plazovi v Sloveniji*. Zaloba ZRC, Ljubljana, 2008.

Interactive As-Rigid-As-Possible Image Deformation and Registration

Marek Dvorožňák*
Supervised by: Daniel Sýkora†

Department of Computer Graphics and Interaction
Faculty of Electrical Engineering
Czech Technical University
Prague / Czech Republic

Abstract

This paper focuses on an existing as-rigid-as-possible deformation model that is particularly suitable for manipulating images that capture articulated objects, for example hand-drawn figures. The model can be used for interactive image deformation as well as automatic image registration. We have implemented both applications as tools for free/open-source image editor GIMP. We describe some details of the implementation and demonstrate functionality of these new tools on a variety of images. For image registration we compare the results of the method with results produced by two existing deformable image registration tools NiftyReg and Drop.

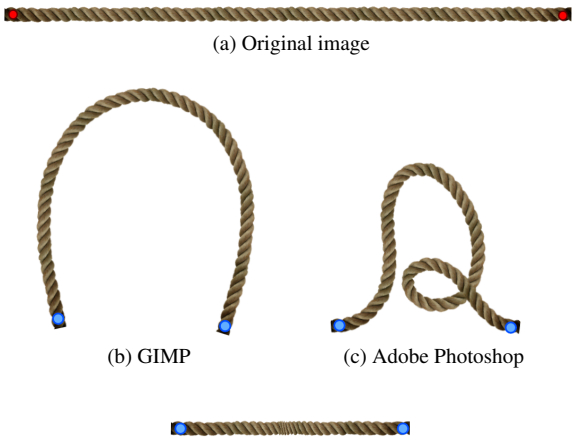
Keywords: image deformation, image registration, as-rigid-as-possible, GIMP

1 Introduction

Image deformation tools implemented in various image editing software allows us to deform image in several ways. Common deformation options include translation, rotation, scaling, shearing and perspective deformation. Recently, various methods have been published which allow user to deform image in a less constrained manner.

In this paper we focus on methods that respect as-rigid-as-possible (ARAP) principle [1]. Its aim is to minimize the amount of local shearing and scaling involved in the deformation. These methods allow user to deform image in a way that during the deformation it behaves like a real world object which is made of rubber.

Some of these ARAP methods are incorporated in recent versions of image editing software. For example, since CS5 version Adobe Photoshop offers Puppet Warp deformation tool based on a method by Igarashi et al. [4], Fiji, a package of tools for image processing, includes Interactive Moving Least Squares deformation plug-in based on Moving Least Squares (MLS) deformation by Schaefer et al. [10] and since 2.7 version graphics editor Krita has Warp tool which employs MLS as well.



(a) Original image

(b) GIMP (c) Adobe Photoshop

(d) Krita

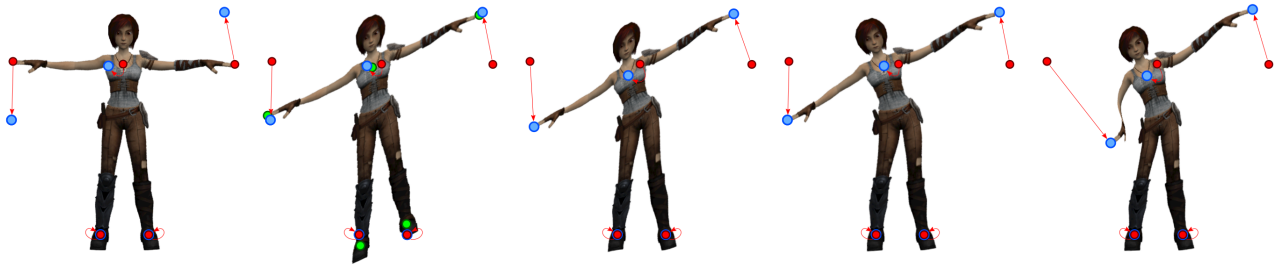
Figure 1: Example of 2-point deformation of a rope using (**b**) our new ARAP deformation tool for GIMP, (**c**) Puppet Warp in Adobe Photoshop and (**d**) Warp tool in Krita.

Free/open-source graphics editor GIMP contains Cage Tool that employs Green Coordinates [5]. The tool allows user to deform image using a cage. The Cage Tool does not preserve as-rigid-as-possible model and thus it is more difficult to obtain realistic deformations of images capturing real world objects. Furthermore, from user point of view, the process of deformation using Cage Tool is relatively cumbersome since at first user has to manually create a cage surrounding the deformed object and only after that he can start deforming the cage using points it is composed of.

As GIMP did not offer an option to deform images in ARAP manner we implemented a tool for ARAP deformation based on a method presented in Wang et al. [13] which allows better deformation results then the aforementioned tools – see Figure 1 for example.

One of the applications of ARAP deformation is ARAP image registration. We extended the ARAP deformation

*dvoromar@fel.cvut.cz
†sykorad@fel.cvut.cz



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

Figure 2: Multipoint deformation of image: (**a**) user specifies initial positions of control points (red) and then moves them onto new positions (blue), (**b**) result of a single affine deformation, (**c**) result of affine MLS deformation, (**d**) result of rigid MLS deformation, (**e**) result of large-scale MLS deformation. The image comes from short computer animated film Sintel.

tool and implemented a tool allowing ARAP registration into GIMP. The implementation is based on an image registration method by Sýkora et al. [12].

The paper is organized as follows. First we focus on some of the methods standing behind ARAP deformation tools and also the ARAP image registration method. Then we describe some details of implementation of the deformation and registration tools into GIMP. Finally we demonstrate functionality of these tools on images of various kinds and compare our image registration results with results produced by two existing deformable image registration tools NiftyReg and Drop.

2 Related Work

As-rigid-as-possible (ARAP) deformation principle was introduced by Alexa et al. [1]. Igarashi et al. [4] later employed this principle in deformation of images capturing articulated objects. They use a triangular mesh respecting boundaries of image. User can fix some mesh vertices and move them onto new locations. Then the following operations are performed: (1) similarity transformation is computed for every triangle and (2) the scaling is removed. According to size of triangles, the deformation need not to be smooth. Schaefer et al. [10] employed Moving Least Squares optimization to produce smooth deformations. They solve an optimization problem for every pixel of image using a closed-form formula which they formulated. However, their method cannot handle large-scale deformations. Sorkine and Alexa [11] formulated ARAP deformation as a non-linear optimization problem and presented how to solve it effectively in iterative manner. Wang et al. [13] used square lattice to compute approximation of the non-linear problem for image deformation. They compute rotations for each square on this lattice using shape matching algorithm proposed by Müller et al. [9] by facilitating the closed-form formula for rotation introduced by Schaefer et al. [10].

In image registration, the goal is to find a deformation (and its parameters) of source image (S) that well aligns it with target image (T). Image registration methods often somehow include *deformation model*, image *similarity measure* and *optimization method*.

There are two basic kinds of image registration methods – feature-based and intensity-based [14]. Feature-based methods use features in source and target image. These features have to be detected and the most correct match of features from one image to the other has to be found. For that purpose SIFT keys [6] are frequently employed. Once we have the features and their correspondences, parameters of a mapping function of a selected deformation model can be found employing the Least Squares method or some other method of parameter estimation [3].

Intensity-based methods work directly with intensities of pixels in image. When source and target image differ only in translation (or also slight rotation), it is possible to determine *globally* optimal shift vector by employing simple block-matching method. However, the method has a high time complexity. Another option is to construct an energetic function $E(\mathbf{t}) = d\big(S(\mathbf{p}+\mathbf{t}),\ T(\mathbf{p})\big)$, where $d(S,T)$ is a dissimilarity function representing dissimilarity measure of images S and T. To obtain a *locally* optimal shift vector $\mathbf{t}$, we can for example employ the gradient descent optimization method [7]. In the same way it is possible to find parameters of mapping function of more complex deformation models. Two of recent representatives of intensity-based methods yielding good results in medical imaging are a method by Glocker et al. [2] and a method by Modat et al. [8]. To solve a problem of registration of hand-drawn images, Sýkora et al. [12] proposed an intensity-based ARAP image registration method that utilizes the deformation method by Wang et al. [13].

3 Image Deformation

In this section we focus on point-based image deformation methods employing ARAP principle. In Figure 2a

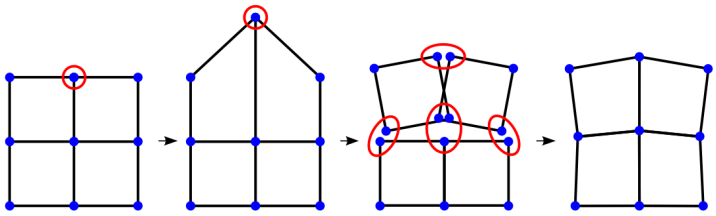


Figure 3: Phases of ARAP deformation of a lattice. Figures (**a**) and (**b**) depict moving one point of the lattice and thus deforming the lattice. Figure (**c**) depicts lattice regularization and (**d**) lattice deformed in as-rigid-as-possible manner. Note that this is just one iteration of the algorithm.

there is an image of a character with control points on it. User first specifies initial positions of these points (indicated by red points) and then he moves these points onto new positions (indicated by blue points) and thus expects to deform the image. Our ultimate goal is to deform the image so that these user specified constraints are satisfied and the amount of local shearing and scaling involved in the deformation is minimized.

3.1 Moving Least Squares Deformation

Our task is to find such affine transformation of coordinates (i.e. transformation matrix $\mathbf{A}$ and translation vector $\mathbf{t}$) which moves red points $\mathbf{p}_i$ so that they are located as close as possible to blue points $\mathbf{q}_i$. Schaefer et al. [10] employed the (Moving) Least Squares method and algebraically formulated the problem as

$$\arg\min_{\mathbf{A},\mathbf{t}} \sum_i w_i \left(\mathbf{A}\mathbf{p}_i + \mathbf{t} - \mathbf{q}_i\right)^2. \tag{1}$$

The weight w_i is defined as

$$w_i = \frac{1}{(\mathbf{p}_i - \mathbf{v})^{2\alpha}}. \tag{2}$$

We can see that it is a function which yields very high values near points $\mathbf{p}_i$. The nearer the pixel $\mathbf{v}$ to some point $\mathbf{p}_j$, the higher is the influence of an affine transformation that maps point $\mathbf{p}_j$ to point $\mathbf{q}_j$; α is a selected parameter.

To simplify the solution of the minimization problem, we first solve for $\mathbf{t}$ and obtain optimal vector of translation $\mathbf{t}_{\min} = \mathbf{q}_c - \mathbf{A}\mathbf{p}_c$, where $\mathbf{q}_c$ and $\mathbf{p}_c$ are weighted centroids of positions of control points, i.e. $\mathbf{p}_c = \frac{\sum_i w_i \mathbf{p}_i}{\sum_i w_i}$, $\mathbf{q}_c = \frac{\sum_i w_i \mathbf{q}_i}{\sum_i w_i}$. After that we can remove the translation from the equation which yields $\arg\min_{\mathbf{A},\mathbf{t}} \sum_i w_i \left(\mathbf{A}\hat{\mathbf{p}}_i - \hat{\mathbf{q}}_i\right)^2$, where $\hat{\mathbf{p}}_i = \mathbf{p}_i - \mathbf{p}_c$, $\hat{\mathbf{q}}_i = \mathbf{q}_i - \mathbf{q}_c$. Then we can solve for $\mathbf{A}$ and obtain optimal transformation matrix $\mathbf{A}_{\min} = \sum_i \left(w_i \hat{\mathbf{q}}_i \hat{\mathbf{p}}_i^T\right) \cdot \left(\sum_i w_i \hat{\mathbf{p}}_i \hat{\mathbf{p}}_i^T\right)^{-1}$.

Figure 2b depicts a result of applying this method on our image with $w_i = 1$. That corresponds to a deformation produced by a single affine transformation. Figure 2c shows a result of the deformation with weights defined as in (2). The result contains an undesirable shearing visible in it.

To obtain a better looking result, it is necessary to extract a rigid transformation out of the affine transformation. For that purpose, matrix decomposition methods such as singular value decomposition or polar decomposition can be employed.

In 2D, closed-form formula that can be used to obtain the rigid transformation directly was presented by Schaefer et al. [10]. Transformation matrix of the rigid transformation is formulated as

$$\mathbf{R}_{\min} = \frac{1}{\mu} \sum_i w_i \begin{pmatrix} \hat{\mathbf{p}}_i \\ -\hat{\mathbf{p}}_i^{\perp} \end{pmatrix} \begin{pmatrix} \hat{\mathbf{q}}_i^T & -\hat{\mathbf{q}}_i^{\perp T} \end{pmatrix} \tag{3}$$

where

$$\mu = \sqrt{\left(\sum_i w_i \mathbf{q}_i \hat{\mathbf{p}}_i^T\right)^2 + \left(\sum_i w_i \mathbf{q}_i \hat{\mathbf{p}}_i^{\perp T}\right)^2} \tag{4}$$

and operator $\perp$ represents a perpendicular vector, i.e. $(x,y)^{\perp} = (y,-x)$.

Figure 2d depicts a result of deformation after processing the image using this as-rigid-as-possible MLS deformation. The result looks more natural.

However, with this approach we only obtain plausible results for suitable positions of points $\mathbf{p}_i$ and $\mathbf{q}_i$. Figure 1d depicts a case where MLS deformation cannot yield desired result because of non-linearity of the problem [11].

Another problem that appears here is caused by the measure (Euclidean distance) that is employed in the weighting function (2). This measure does not respect a topology of image. The consequence of this is that when we e.g. deform an image of a character and we move a point located on a hand towards the body, not only the hand is deformed but also hips and the body (see Figure 2e). This can be solved by a measure that respects image topology.

3.2 Rigid Square Matching

A method that does not suffer from the aforementioned problems was presented by Wang et al. [13]. Examples of

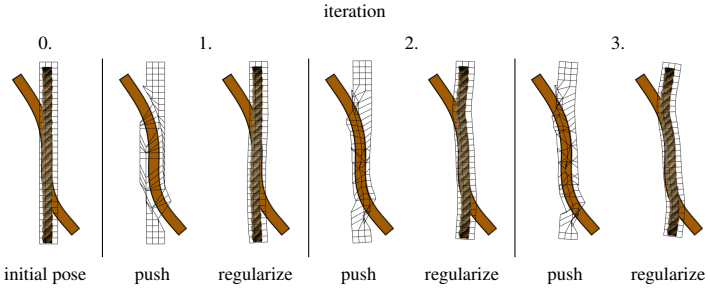


Figure 4: Several first iterations of ARAP image registration algorithm. The goal is to deform the image of a rope to well align it with the image of bent rectangle.

image deformations produced by this method are depicted in Figure 5.

In this method a square lattice (mesh) copying shape of image is firstly constructed above image. Some points of the lattice are moved by user to desired locations and the lattice is then iteratively deformed in as-rigid-as-possible manner. Deformed image is obtained by using any of suitable 4-point image deformation methods on each square of the lattice.

The mesh composed of squares is used despite the fact that with it we are not able to precisely copy the image (as with triangular mesh). An advantage of square mesh is simplicity of its construction (in comparison to e.g. triangulation) and a lower number of lattice elements in comparison to triangular mesh. A shape-aware mesh splitting algorithm [13] can be employed to achieve similar behaviour as with triangular mesh.

A lattice is actually a group of points. Similarly to section 3.1 we have initial positions $\mathbf{p}_i$ of points of the lattice (or alternatively *source* lattice) and target positions $\mathbf{q}_i$ of the lattice (or *target* lattice). User sets new positions of some of the points $\mathbf{q}_i$ of the target lattice (Figure 3a) and thus deforms the lattice – some of the lattice squares become quadrilaterals (see Figure 3b).

The core of the method is in a regularization phase where we try to respect user specified constraints in a form of the placed points $\mathbf{q}_i$ as well as deform the lattice in a way that every single square (quadrilateral) is deformed the most rigidly. We perform a specified number of regularizing iterations of which every one performs the following operations:

1. Rigid transformation of every lattice square (see Figure 3c) using the formula (3) and $w_i = 1$. This transforms every quadrilateral to square again.

2. Centering every originally overlapping points of the lattice (see Figure 3d). This ensures that points in every cluster of originally overlapping points of the lattice will overlap again – and squares become quadrilaterals again.

Hundreds of these iterations are usually performed. Convergence of the method is slower than with ARAP deformation methods that solve linear system [11] instead of computing centroids. However, the advantage of the method is that it does not require us to specify fixed points. This fact is utilized in image registration [12].

4 ARAP Image Registration

In this section we will focus on an application of ARAP image deformation. We will describe an intensity-based registration method presented by Sýkora et al. [12]. The method was invented for usage in cartoon industry and is especially suitable for registration of hand-drawn articulated images.

When registering such images, it is not possible to use methods based on features since every drawing is unique to some extent and hence it is not possible to find corresponding features. In such a case methods based on optimization of energetic function may lead to a success. However, if source and target image differ in large non-linear deformation, these methods usually get stuck in a local extrema and hence the result of these methods will often not be plausible – see Figure 7f.

The method by Sýkora et al. employs the rigid square matching deformation algorithm described in section 3.2, sum of absolute differences (SAD) dissimilarity measure and block-matching optimization method. The method takes advantage of the fact that the deformation algorithm allows us to arbitrarily move points $\mathbf{q}_i$ without having to consider their mutual connection. The actual registration is divided into "push" and "regularize" phases. These two phases are continuously performed until fulfillment of a stop condition.

In Figure 4 there are two overlapping objects depicted – image of a rope (source image) and image of a bent rectangle (target image). The aim of this registration problem is to deform the image of a rope so that it well aligns with the image of a bent rectangle.

In the *push* phase of ARAP registration, we move lattice points $\mathbf{q}_i$ to locations where the area of the source image around these points differs *as little as possible* from the area of the target image (around these points). To find such a translation vector $\mathbf{t}$ the method employs the block-matching method that finds the optimal translation

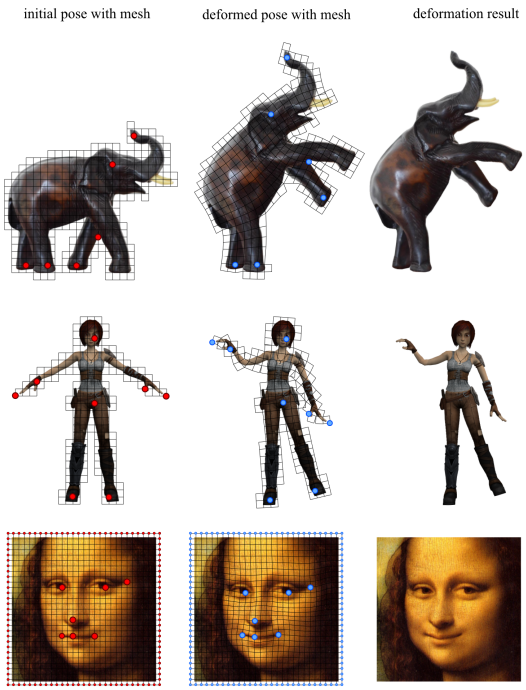


Figure 5: Examples of image deformation created using our N-Point Deformation tool in GIMP. Red points indicate initial positions of points that are *fixed*, blue points indicate desired positions of these points.

vector in defined search area. For SAD dissimilarity measure the optimal translation vector $\mathbf{t}_{\text{opt}}$ can be formulated as

$$\mathbf{t}_{\text{opt}} = \arg\min_{\mathbf{t}\in M} \sum_{\mathbf{p}\in N} |S(\mathbf{p}+\mathbf{t}) - T(\mathbf{p})|, \qquad (5)$$

where M is a search area where we search for optimal shift, N is SAD "neighborhood", S is source and T is target image. In *regularization* phase the lattice is put into a consistent state by the rigid square matching algorithm. Figure 4 shows several first iterations of ARAP image registration algorithm together with results of push and regularize phases for each iteration.

By the number regularizing iterations in the rigid square matching algorithm, we can adjust rigidity (or flexibility) of deformation. That is utilized in this image registration method to refine the result – see Figure 7, 8, 9, 10.

5 Implementation

Our goal was to implement the rigid square matching algorithm and the ARAP image registration algorithm into GIMP. For implementation C programming language was selected since GIMP is written in this language.

We implemented (1) the rigid square matching algorithm into a new library named *libnpd*, (2) an operation that allows ARAP image deformation into GEGL library which is employed in GIMP; the operation utilizes libnpd library, (3) ARAP image deformation tool into GIMP; the tool utilizies the aforementioned operation, (4) ARAP image registration tool into GIMP; the tool extends the deformation tool.

In this paper, we use the term "n-point deformation" for image deformation employing the rigid square matching algorithm and the term "n-point registration" for image registration employing the ARAP image registration algorithm.

5.1 N-Point Deformation Library (libnpd)

The library contains data types and functions allowing to perform n-point image deformation and registration. The library is designed in a way so that it can be used with various graphics libraries and thus requires to implement some graphics functions (e.g. get_pixel_color, set_pixel_color) and data types (image, display).

One of the most important data types in the library is NPDModel type which holds source lattice, target lattice (i.e. "source" and "target" group of points), source image and display.

5.2 GIMP and GEGL Library

As already stated, GIMP is written in C (C89 standard). The C language itself is not object-oriented. GIMP uses the C language enriched with object-oriented approach using GObject object system, which is part of GLib library.

GIMP currently uses GEGL and BABL libraries which should allow it to work with high depth color images and use some non-destructive image editing techniques. GEGL is a graphics library which uses specified oriented acyclic graph to process images. This graph is made up of individual nodes that can represent graphics operations as well as another graph. Edges that connect individual nodes determine the order in which the graph will be processed. Nodes usually expect image on their inputs and produce (filtered) image on their output.

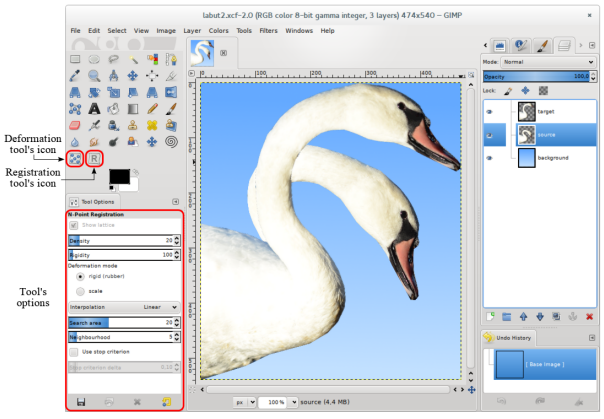


Figure 6: Our ARAP image deformation and registration tools integrated into the environment of GIMP

(a) source image (b) target image (c) initial overlap (d) NPR_1 (e) NPR_2 (f) Drop (g) NiftyReg

Figure 7: Registration of a hand-drawn images. The task is to deform the source image to well align it with the target image. Resolution of images: 420×541. NPR_1 is a result of NPR tool with rigidity set to 600, for NPR_2 the rigidity is 30. Courtesy of Universal Production Partners.

(a) source image (b) target image (c) initial overlap (d) NPR_1 (e) NPR_2 (f) Drop (g) NiftyReg

Figure 8: Registration of a hand-drawn images. Resolution of images: 560×400. NPR_1 is a result of NPR tool with rigidity set to 200, for NPR_2 the rigidity is 30. Courtesy of AniFilm.

Functionality of GIMP can be extended using plug-ins. They can be written in various programming languages – namely Scheme, Python, Perl and C. GIMP incorporates a large amount of plug-ins, most of them function as graphic filters. The current development trend is not to create graphic filters in form of GIMP plug-ins but instead to create GEGL operations within GEGL library.

N-point deformation algorithm was implemented into the GEGL library as a new operation. This operation employs libnpd library. There can be two scenarios of its usage depending on whether we *have* or we *do not have* NPDModel.

When we do not have NPDModel, the operation expects image on its input. After the first processing of a graph the operation is contained in, the operation creates NPDModel and returns it through operation's parameters. User can then deform the target lattice contained in NPDModel by manipulating with lattice's points. Second and another processing produces ARAP deformed image that is result of a specified number of deformation iterations. This image is available on operation's output.

When we have NPDModel, we can supply it to the operation through operation's parameter and perform the deformation in the same way as described in previous paragraph.

5.3 N-Point Deformation Tool

N-point deformation was implemented as internal tool (instead of a plug-in). That allows the tool's GUI to be seamlessly integrated in GIMP. Individual control points can be placed directly on the canvas. During the deformation process, user can use GIMP's GUI e.g. to easily zoom to a certain part of image that is being deformed and focus on details or he can arbitrarily rotate the canvas.

Every tool in GIMP is implemented as a class extending a parent class named GimpTool. This parent class provides the basic functionality common to all tools in GIMP. This class is extended by a class named GimpDrawTool that allows its descendant classes to add GUI elements on canvas and that is able to draw these elements. These elements include control points (handles), basic plane shapes, guide lines, paths, text cursor and also a live preview of a result of operation that is performed by a tool. This class is extended by various classes representing tools, let us mention for example a group of painting tools, selection tools, transformation tools and also a tool for writing text.

The class GimpDrawTool is also extended by our class named GimpNPointDeformationTool that implements the N-Point Deformation tool. This class employs libnpd library. Using a deformation thread, it performs a deformation of image, using a preview thread, it draws at regular intervals a preview of current state of the deformation. The preview thread calls the methods of GimpDrawTool in order to redraw GIMP's GUI.

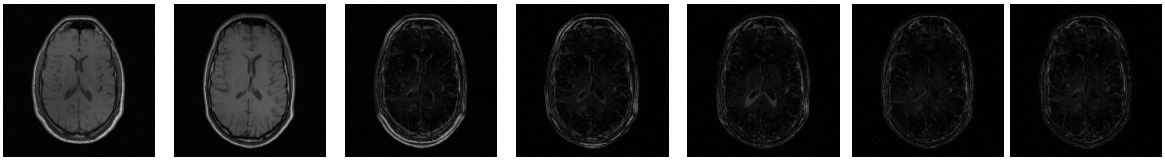
Every GIMP tool can define its own set of settings and their graphic representation within GIMP tool's GUI. For these purposes, a class named GimpToolOptions is employed. This class is inherited by classes describing individual tools options. N-Point Deformation tool employs its own class named GimpNPointDeformationOptions.

5.4 N-Point Registration Tool

As the *n*-point deformation, the *n*-point registration was implemented into GIMP as an internal tool. The main rea-

(a) source image (b) target image (c) initial overlap (d) NPR_1 (e) NPR_2 (f) Drop (g) NiftyReg

Figure 9: Registration of images of a person. Resolution of images: 489×656. NPR_1 is a result of NPR tool with rigidity set to 200, for NPR_2 the rigidity is 30.



(a) source image (b) target image (c) initial overlap (d) NPR_1 (e) NPR_2 (f) Drop (g) NiftyReg

Figure 10: Registration of images of a brain. Resolution of images: 421×436. NPR_1 is a result of NPR tool with rigidity set to 50, for NPR_2 the rigidity is 5. Difference images have been used to show the resulting overlap. Images come from the RIRE dataset.

sons for this decision included the possibility to use existing implementation of N-Point Deformation tool and the possibility to allow user to easily help the registration (by moving one or several points to correct locations) in situation when the registration get stuck in undesirable state.

A class named GimpNPointRegistrationTool which extends GimpNPointDeformationTool class was created. The N-Point Deformation tool class was modified to allow its employment for registration purposes. As previously mentioned, the N-Point Deformation tool uses a thread to perform the deformation. Within this thread a method performing the deformation is being called. This method is overridden in GimpNPointRegistrationTool by a method performing the ARAP image registration algorithm.

A class GimpNPointRegistrationOptions defining a set of settings of the tool was created. This class extends GimpNPointDeformationOptions class which was modified to allow its subclasses to use a set of generic settings of the deformation.

6 Results

We used our N-Point Deformation tool to deform several images – see Figure 1 and 5. Let us look at Figure 1 depicting a deformation of a rope using various deformation tools. Krita (and Fiji) produced undesirable result. With Adobe Photoshop we can obtain a result similar to ours, however, the process is cumbersome. With Puppet Warp tool user must rotate control points (pins) to achieve desired deformation. There is an automatic pin rotating function available, however, as in our example it might not work correctly. There is no need to rotate control points in N-Point Deformation tool. Thanks to the method employed in our tool, the deformation is predictable and the deformation process is easier than with Puppet Warp in Adobe Photoshop.

We compared our N-Point Registration (NPR) tool with deformable image registration tools Drop [2] and NiftyReg [8]. Figure 7 shows a plausible result of NPR tool. Even we tried to set the best parameters in Drop and NiftyReg tools we obtained unsatisfactory results. Here, the deformation model preserving rigidity is a great advantage. Figure 8 shows an example where all three tools gave satisfactory result. There we can see that Drop and NiftyReg produced results that look more similar to the target image. However, the details are more distorted. Figure 9 shows registration of images of a person. In this problem we obtained plausible results with NPR and Drop. The latter produced really good result, however, there is again problem with details (left hand). With NiftyReg we could not obtain satisfactory result again. In a registration problem involving registration of articulated images, good approach is to first perform ARAP registration and then refine with a non-linear registration method as in Drop or NiftyReg. Sýkora et al. use that approach in [12].

Figure 10 shows results of a common registration problem – aligning two medical images. Drop and NiftyReg were designed exactly for this kind of a problem and gave very good results. We can see that NPR can also handle this kind of registration problem because the rigid square matching algorithm allows some amount of shrinking or stretching (depending on a number of deformation iterations). However, the result is not as good as with the two other tools.

7 Conclusions and Future Work

We implemented as-rigid-as-possible image deformation and registration tools into a development version of popular free/open-source image editor GIMP and demonstrated their functionality on images of various kinds.

For image registration it is evident from the results that some modern non-linear image registration methods, when properly set, are able to cope even with a large deformation of images entering the registration. In contrast to these methods in some situations a great advantage of the ARAP image registration method can be the fact that in almost all circumstances it produces results that are not unnaturally deformed. This is particularly useful when registering real or cartoon figures and their poses.

Although the tools are functional they still have some weaknesses. Further work is thus to eliminate them. The tools have currently problems regarding speed when working with large images, mainly due to redrawing the preview of the deformation. The preview has to be rendered several times per second, which is what causes the problems. In the registration tool the problem is also caused by block-matching method that is employed during registration. For large images, it is necessary to set a higher value of the search parameters ("search area" and "neighborhood"). Currently, user does not have an option to set a depth of individual control points and thus he cannot specify which part of the overlapping lattice (image) will be visible. The tools currently do not use multiple CPU/GPU cores to speed up their computations.

Acknowledgements

Part of the work on N-Point Deformation tool has been supported by Google through Google Summer of Code 2013. Thank must go to GIMP developers for selecting the project.

This work has been partially supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS13/214/OHK3/3T/13 (Research of Progressive Computer Graphics Methods).

References

[1] Marc Alexa, Daniel Cohen-Or, and David Levin. As-rigid-as-possible shape interpolation. In *ACM SIGGRAPH Conference Proceedings*, pages 157–164, 2000.

[2] Ben Glocker, Nikos Komodakis, Georgios Tziritas, Nassir Navab, and Nikos Paragios. Dense image registration through MRFs and efficient linear programming. *Medical Image Analysis*, 12(6):731–741, 2008.

[3] A. Ardeshir Goshtasby. Robust parameter estimation. In *Image Registration*, Advances in Computer Vision and Pattern Recognition, pages 313–341. 2012.

[4] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. As-rigid-as-possible shape manipulation. *ACM Transactions on Graphics*, 24(3):1134–1141, 2005.

[5] Yaron Lipman, David Levin, and Daniel Cohen-Or. Green coordinates. *ACM Transactions on Graphics*, 27(3):78, 2008.

[6] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[7] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, volume 2, pages 674–679, 1981.

[8] Marc Modat, Gerard R. Ridgway, Zeike A. Taylor, Manja Lehmann, Josephine Barnes, David J. Hawkes, Nick C. Fox, and Sébastien Ourselin. Fast free-form deformation using graphics processing units. *Computer Methods and Programs in Biomedicine*, 98(3):278–284, 2010.

[9] Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Meshless deformations based on shape matching. *ACM Transactions on Graphics*, 24(3):471–478, 2005.

[10] Scott Schaefer, Travis McPhail, and Joe Warren. Image deformation using moving least squares. *ACM Transactions on Graphics*, 25(3):533–540, 2006.

[11] Olga Sorkine and Marc Alexa. As-rigid-as-possible surface modeling. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*, pages 109–116, 2007.

[12] Daniel Sýkora, John Dingliana, and Steven Collins. As-rigid-as-possible image registration for hand-drawn cartoon animations. In *Proceedings of International Symposium on Non-photorealistic Animation and Rendering*, pages 25–33, 2009.

[13] Yanzhen Wang, Kai Xu, Yueshan Xiong, and Zhi-Quan Cheng. 2D shape deformation based on rigid square matching. *Computer Animation and Virtual Worlds*, 19(3–4):411–420, 2008.

[14] Barbara Zitová and Jan Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21(11):977–1000, 2003.

Lighting and Natural Phenomena

Hatching for Metaball Surfaces

Ferenc Tükör
Supervised by: László Szécsi[*]

Institute of Computer Graphics
Budapest University of Technology and Economics
Budapest / Hungary

Abstract

This paper presents a highly parallel algorithm for the stylized, real-time display of fluids and smoke. We use metaballs to define a fluid surface from a particle-based fluid representation, but instead of the costly complete reconstruction of this surface, we only trace the motion of random seed points on it. Hatching strokes are extruded along the lines of curvature. We propose methods for hidden stroke removal and density control that maintain animation consistency.

Keywords: Hatching, NPR, Metaball

1 Introduction

Hatching is an artistic technique that is often emulated in stylistic animation. Implicit surfaces are becoming increasingly important in real-time applications for visualizing fluids simulated by particle-based methods. Thus, we aim to extend real-time hatching to deforming implicit surfaces, and specifically to metaballs. In addition to enabling fluid rendering in hatching-style NPR, we also consider this approach a more feasible alternative to expensive polygonization [12, 18], ray casting [15, 3], or screen-space filtering [20], when visualizing scientific isosurface or fluid simulation data.

2 Previous work

In pencil drawings artists convey the shape and illumination of objects with the density and orientation of thin hatch lines [23, 7]. To mimic this, we should define a *density* and a *direction field* in the image plane that is as close as possible to what an artist would use. Density is influenced by illumination, while the direction field is determined either by the *principal curvature directions* [6], the tone gradient [10], or in case of animation, the direction of motion.

Hatch strokes should appear hand-drawn, with roughly similar image-space width, dictated by brush size, but they should also stick to surfaces to provide proper object space shape and motion cues. Hatches can be generated into textures and mapped onto animated objects, with level-of-detail mechanisms to approximate image space behavior [17]. In absence of surface parametrization, this approach is not applicable to implicit surfaces.

Hatch strokes can be generated directly in image space [9, 11], but if they are fixed in their position and cling to the view plane instead of the animated objects, movement will appear as if seen through textured glass. This is known as the *shower door effect*. In order to avoid this disturbing phenomenon, lines can be moved along with an optical flow or image space velocity field, but placing new strokes on emerging, previously non-visible surfaces still poses problems. Especially if strokes are long, following curvature or feature curves of object surfaces, they should appear consistent even when only tiny fractions have become visible. This cannot be assured when only using image space information. For implicit surfaces, it is often prohibitively expensive to render a full image, or even to find isosurfaces in some pixels.

Several works [13, 19] proposed the application of *particles* or *seeds* attached to objects, extruding them to hatch strokes in image space. The key challenge in these methods is the generation of the world-space seed distribution corresponding to the desired image-space hatching density. This approach is well-suited to implicit surfaces.

Much effort was directed at rendering implicit surfaces, esp. metaballs photorealistically in real time, based on ray casting [8]. This is computationally demanding as it requires a high number of field function evaluations to find the visible isosurface in every pixel. The stylization of the result is straightforward only with image-space techniques, as no surface parametrization or visibility-independent object-space shape information is extracted.

Several aspects of stylized rendering of implicit surfaces have been studied. Brazil et al. [22] use *seed points* to generate *render points* on isosurfaces. They require the user to edit seed point distribution manually, excluding application for deforming surfaces. Elber [5] proposed the approach of first obtaining a Euclidean-space on-surface uniform point distribution, then extruding strokes along symbolically computed principal curvature directions [26, 16]. For the generation of uniformly distributed points on implicit surfaces, they refer the reader to Witkin [24], who

[*]szecsi.laszlo@gmail.com

proposes an adaptive resampling of deformed implicit surfaces, for purposes of sculpting, by the means of *repulsion forces*, *fissioning* and *killing* operating on a set of *floater particles*. Kooten et al. [21] employ a similar concept more specifically for isosurface rendering of metaball models. Both solutions require a full *self-spatial join* on surface particles to compute repulsion forces, and allow particles to float on surfaces. We consider this detrimental for our purpose of hatching stylization, as hatch lines not moving with the surface could provide inappropriate motion cues. A rejection-based density control approach from [19] does not require repulsion forces to achieve desired distribution.

3 Implicit surfaces and metaballs

An implicit surface is defined as an isosurface at value L of field function $f(\mathbf{x})$ with the implicit equation $f(\mathbf{x}) = L$. Its gradient $\mathbf{g}(\mathbf{x})$ is $\nabla f(\mathbf{x})$.

Metaballs [14, 2] constitute a special case where the fluid is represented by a number of balls or *atoms* as

$$f(\mathbf{x}) = \sum_{j=0}^{M-1} f_j(\mathbf{x}) = \sum_{j=0}^{M-1} \rho_j\left(\left\|\mathbf{x}-\mathbf{a}_j\right\|\right), \qquad (1)$$

with M as the number of atoms, ρ_j the generator of *radial basis function* $f_j(\mathbf{x})$ for an *atom* centered at $\mathbf{a}_j$. If $\rho_j(r)$ has finite support, i.e. $\exists R_j \in \mathbb{R} : \forall r > R_j : \rho_j(r) = 0$, then we call R_j the *effective radius* of atom j.

The gradient $\mathbf{g}(\mathbf{x})$ and Hessian $\mathbf{H}(\mathbf{x})$ can be computed as sums of atom gradients $\mathbf{g}_j(\mathbf{x})$ and atom Hessians $\mathbf{H}_j(\mathbf{x})$.

Gaussian and mean curvatures K and H, the principal curvatures κ_1 and κ_2, principal curvature directions $\mathbf{t}_1, \mathbf{t}_2$ can be computed [1] using the Hessian $\mathbf{H}(\mathbf{x})$. Where the determinant $D = H^2 - K$ is close to zero, the principal curvatures are not well defined, and we regard the surface point as umbilical.

The approach we employ extrudes textured triangle strips along a metaball surface, in the principal curvature directions. The method we use to move seeds along a metaball surface is similar to [21]. First we cover formula derivations for popular radial basis functions to get the above-mentioned, necessary quantities then we describe the details of curvature computation.

3.1 Gradients and Hessians

In order to be able to evaluate the curvature formulae, we need to compute the field function, its gradient, and Hessian. Those are all obtained as the sum of respective functions for metaball atoms. Here we give the formulae for the infinite support *Blinn* [2] and finite support *Wywill* [25] functions. We give all base functions, gradients and Hessians as functions of $\mathbf{d} = \mathbf{x} - \mathbf{a}$, where $\mathbf{a}$ is the atom position. This is to avoid having to subtract $\mathbf{a}$ at every instance of $\mathbf{x}$.

Before the derivations let us introduce the vectors of pure and mixed second-order partial derivatives as

$$\mathbf{p} = \begin{bmatrix} \frac{\partial^2 \mathbf{f}}{\partial x^2} & \frac{\partial^2 \mathbf{f}}{\partial y^2} & \frac{\partial^2 \mathbf{f}}{\partial z^2} \end{bmatrix}^{\mathrm{T}}$$

and

$$\mathbf{m} = \begin{bmatrix} \frac{\partial^2 \mathbf{f}}{\partial x \partial y} & \frac{\partial^2 \mathbf{f}}{\partial y \partial z} & \frac{\partial^2 \mathbf{f}}{\partial z \partial x} \end{bmatrix}^{\mathrm{T}}.$$

With these Hessian is

$$\mathbf{H} = \begin{bmatrix} p_x & m_x & m_z \\ m_x & p_y & m_y \\ m_z & m_y & p_z \end{bmatrix}.$$

The Blinn base function is:

$$f^{\mathrm{Blinn}}(\mathbf{d}) = \frac{1}{\|\mathbf{d}\|^2}.$$

The gradient is:

$$\mathbf{g}^{\mathrm{Blinn}}(\mathbf{d}) = -\mathbf{d}\frac{2}{\|\mathbf{d}\|^4}.$$

Let us introduce the notation for a *swizzle* of a vector $\mathbf{y}$

$$\mathbf{y}_{yxz} = \begin{bmatrix} y_y \\ y_x \\ y_z \end{bmatrix},$$

and similarly for any order of elements. With this the vector of pure second derivatives $\mathbf{p}(\mathbf{d})$, using $\mathbf{e} = \mathbf{d} \circ \mathbf{d}$, where $\circ$ is the Hadamard product operator, is:

$$\mathbf{p}^{\mathrm{Blinn}}(\mathbf{d}) = \frac{6\mathbf{e} - 2\left(\mathbf{e}_{yzx} + \mathbf{e}_{zxy}\right)}{\|\mathbf{d}\|^6}.$$

The vector of mixed second derivatives $\mathbf{m}(\mathbf{d})$ is

$$\mathbf{m}^{\mathrm{Blinn}}(\mathbf{d}) = \frac{8\mathbf{d} \circ \mathbf{d}_{yzx}}{\|\mathbf{d}\|^6}.$$

The Wywill base function has finite support. Let R be the effective atom radius, and introduce the shorthand $\delta = \|\mathbf{d}\|/R$. With these, the Wywill base function is:

$$f^{\mathrm{Wywill}}(\mathbf{d}) = \begin{cases} 0 & \text{if } \delta > 1, \\ 1 + \frac{-4\delta^6 + 17\delta^4 - 22\delta^2}{9} & \text{if } \delta \leq 1. \end{cases} \qquad (2)$$

With

$$G = \frac{4\left(6\delta^4 - 17\delta^2 + 11\right)}{9R^2},$$

the gradient is:

$$\mathbf{g}^{\mathrm{Wywill}}(\mathbf{d}) = \begin{cases} \mathbf{0} & \text{if } \delta > 1, \\ -\mathbf{d}G & \text{if } \delta \leq 1. \end{cases}$$

The vector of pure second derivatives $\mathbf{p}(\mathbf{d})$, using $\mathbf{e} = \mathbf{d} \circ \mathbf{d}$ is:

$$\mathbf{p}^{\mathrm{Wywill}}(\mathbf{d}) = \begin{cases} \mathbf{0} & \text{if } \delta > 1, \\ \frac{4\mathbf{e}\left(17 - 12\delta^2\right)}{R^4} - \begin{bmatrix} G \\ G \\ G \end{bmatrix} & \text{if } \delta \leq 1. \end{cases}$$

The vector of mixed second derivatives $\mathbf{m}(\mathbf{d})$ is:

$$\mathbf{m}^{\text{Wywill}}(\mathbf{d}) = \begin{cases} \mathbf{0} & \text{if } \delta > 1, \\ \mathbf{d}_{xyz} \circ \mathbf{d}_{yzx} \frac{4\left(12\delta^2 - 17\right)}{9R^4} & \text{if } \delta \leq 1. \end{cases}$$

3.2 Curvature computation

Here we continue using the notations for the vectors of pure and mixed second order partial derivatives and the Hessian from Section 3.1.

The following method of curvature computation is based on [1]. All quantities are functions of $\mathbf{x}$, which we will omit in the notation for ease of reading.

The Gaussian curvature K is

$$K = -\frac{1}{\|\mathbf{g}\|^4} \begin{vmatrix} \mathbf{H} & \mathbf{g} \\ \mathbf{g}^{\mathrm{T}} & 0 \end{vmatrix}. \tag{3}$$

With normal $\mathbf{n} = -\frac{\mathbf{g}}{\|\mathbf{g}\|}$, and Laplacian $\Delta\mathbf{f} = \frac{\partial^2 \mathbf{f}}{\partial x^2} + \frac{\partial^2 \mathbf{f}}{\partial y^2} + \frac{\partial^2 \mathbf{f}}{\partial z^2}$ the mean curvature H is

$$H = \frac{1}{\|\mathbf{g}\|} \left[\mathbf{n}^{\mathrm{T}} \mathbf{H} \mathbf{n} - \Delta\mathbf{f}\right].$$

The principal curvatures are:

$$\begin{aligned} \kappa_1 &= H + \sqrt{(H)^2 - K}, \\ \kappa_2 &= H - \sqrt{(H)^2 - K}. \end{aligned} \tag{4}$$

We need to construct the matrix

$$\left(\mathbf{n} \cdot \mathbf{n}^{\mathrm{T}} - \mathbf{I}\right) \mathbf{H} - \mathbf{I} \kappa_1 \|\mathbf{g}\|,$$

where $\mathbf{I}$ is the identity matrix, then take the maximum length one out of the three possible pairwise cross products of its rows. Normalized, it gives principal direction $\mathbf{t}_1$. Then, $\mathbf{t}_2 = \mathbf{t}_1 \times \mathbf{n}$.

Using the *swizzle* notation from Section 3.1 the determinant of equation 3 can be computed without explicitly constructing the matrix as

$$\begin{aligned} \begin{vmatrix} \mathbf{H} & \mathbf{g} \\ \mathbf{g}^{\mathrm{T}} & 0 \end{vmatrix} = \\ & 2(\mathbf{p} \circ \mathbf{m}_{yzx}) \cdot (\mathbf{g}_{yzx} \circ \mathbf{g}_{zxy}) \\ & - (\mathbf{p}_{zxy} \circ \mathbf{p}_{yzx}) \cdot (\mathbf{g} \circ \mathbf{g}) \\ & + (\mathbf{m} \circ \mathbf{m}) \cdot (\mathbf{g}_{zxy} \circ \mathbf{g}_{zxy}) \\ & - 2(\mathbf{m}_{xzy} \circ \mathbf{m}_{yxz}) \cdot (\mathbf{g}_{xzy} \circ \mathbf{g}_{zyx}). \end{aligned} \tag{5}$$

4 Seeds and their motion explained

Seeds are particles moving along the deforming isosurface. The velocity vector used to move a seed is found by using the formulae described in [21]. There are three effects that contribute to this motion: fluid motion, field shift, and correction.

4.1 Complete seed velocity

4.2 Fluid motion

The fluid medium itself is moving. Its motion is defined for atoms with atom velocities $\mathbf{q}_j$. How we construct the flow velocity at a point from these relies on the requirement that points on the isosurface should remain on the isosurface. How much the linear motion of an atom influences the isosurface depends on the length of the base function gradient at the isosurface point. Thus, linear atom velocities should be weighted with this gradient length to get the flow velocity:

$$\mathbf{v}^{\mathrm{fl}}(\mathbf{s}) = \frac{\sum_{j=0}^{M-1} \left\|\mathbf{g}_j(\mathbf{s}_k)\right\| \mathbf{q}_j}{\sum_{j=0}^{M-1} \left\|\mathbf{g}_j(\mathbf{s}_k)\right\|}.$$

The seeds need to travel along the isosurface, so the fluid velocity must be projected on it. The component perpendicular to the surface is found as

$$\mathbf{v}_k^{\mathrm{perp}} = \frac{\mathbf{g}(\mathbf{s}_k)\left(\mathbf{v}_k^{\mathrm{fl}} \cdot \mathbf{g}(\mathbf{s}_k)\right)}{\|\mathbf{g}(\mathbf{s}_k)\|^2},$$

and thus the projected fluid velocity is

$$\mathbf{v}_k^{\mathrm{pfl}} = \mathbf{v}_k^{\mathrm{fl}} - \mathbf{v}_k^{\mathrm{perp}} = \mathbf{v}_k^{\mathrm{fl}} - \frac{\mathbf{g}(\mathbf{s}_k)(\mathbf{v}_k^{\mathrm{fl}} \cdot \mathbf{g}(\mathbf{s}_k))}{\|\mathbf{g}(\mathbf{s}_k)\|^2}.$$

4.3 Surface pull

Seeds need to move towards the isosurface either because they are distant due to initial or accumulated error, or because the isosurface itself has moved. For both effects, we will be able to find the desired rate of change in field value at the seed $\delta = \frac{\partial f(\mathbf{s}_k)}{\partial t}$, and need to compute the seed velocity $\mathbf{v}_k^{\mathrm{pull}} = \partial \mathbf{s}_k / \partial t$ from this. We move the seed along the gradient, so $\mathbf{v}_k^{\mathrm{pull}} = \xi \mathbf{g}(\mathbf{s}_k)$ with some ξ. It must be true that

$$\delta = (\xi \mathbf{g}(\mathbf{s}_k)) \cdot \mathbf{g}(\mathbf{s}_k).$$

Solving this for ξ gives

$$\xi = \frac{\delta}{\|\mathbf{g}(\mathbf{s}_k)\|^2},$$

and then

$$\mathbf{v}_k^{\mathrm{pull}} = \frac{\mathbf{g}(\mathbf{s}_k)\delta}{\|\mathbf{g}(\mathbf{s}_k)\|^2}.$$

Correction

As neither the temporal nor the spatial linearizations applied are accurate, the seeds positions would accumulate error and drift away from the isosurface. Also, when initialized, the seeds are at random positions and need to be drawn to the isosurface rapidly. Therefore, a correction term with boldness factor Φ is applied. The boldness factor Φ is the inverse of the time in which the seed is supposed to reach the isosurface. Thus, δ^{corr} is $(L - f(\mathbf{s_k}))\Phi$.

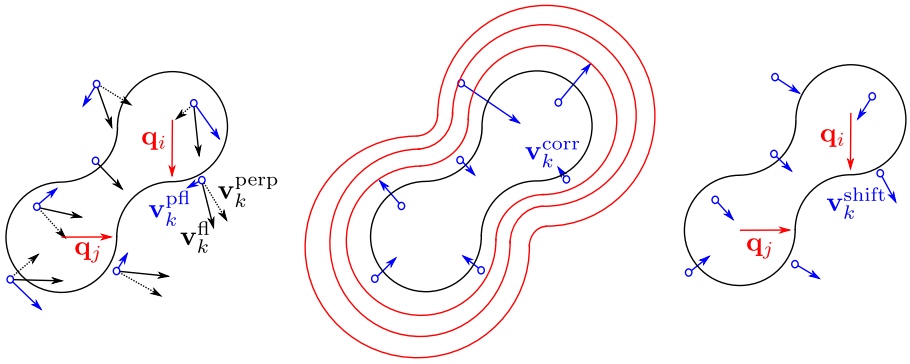


Figure 1: The components of seed velocity: the surface-projected fluid velocity (*left*), the correction term toward the surface along the gradient (*center*), and the term following the isosurface shift due to chaging field values (*right*).

However, large Φ values can lead to instabilities near strongly non-linear regions of the field function.

$$\mathbf{v}_k^{\text{corr}} = \frac{\mathbf{g}(\mathbf{s_k})(L - f(\mathbf{s_k}))\,\Phi}{\|\mathbf{g}(\mathbf{s_k})\|^2}.$$

Field shift

As atoms move, the field value at a $\mathbf{s}_k$ is going to increase or decrease. This change will make the isosurface of L move along the gradient. The rate of change at seed k due to atom j moving is:

$$\delta_j^{\text{shift}} = -\mathbf{g}_j(\mathbf{s}_k) \cdot \mathbf{q}_j,$$

and the total effect of all atoms is:

$$\delta^{\text{shift}} = -\sum_{j=0}^{M-1} \mathbf{g}_j(\mathbf{s}_k) \cdot \mathbf{q}_j.$$

This makes the shift velocity:

$$\mathbf{v}_k^{\text{shift}} = -\frac{\mathbf{g}\sum_{j=0}^{M-1} \mathbf{g}_j(\mathbf{s}_k) \cdot \mathbf{q}_j}{\|\mathbf{g}\|^2}.$$

All terms, save for the unprojected fluid velocity, contain the gradient divided by its length squared. Their sum can therefore be written as:

$$\mathbf{v}_k = \mathbf{v}_k^{\text{fl}} - \tag{6}$$

$$\frac{\mathbf{g}(\mathbf{s_k})}{\|\mathbf{g}(\mathbf{s_k})\|^2}\left(\mathbf{v}_k^{\text{fl}} \cdot \mathbf{g}(\mathbf{s}_k) + (f(\mathbf{s_k}) - L)\,\Phi + \sum_{j=0}^{M-1} \mathbf{g}_j(\mathbf{s}_k) \cdot \mathbf{q}_j\right)$$

A visual representation of this equation can be seen in Figure 1.

5 The algorithm

Our algorithm moves seeds along a metaball surface similar to [21], applies a screen-space approximate version of the density control approach from [19], and extrudes textured triangle strips along principal curvature directions.

We propose a solution for the seed visibility problem based on the idea employed by *variance shadow maps* [4]. The algorithm performs the following steps in every frame of an animation:

1. Initialization of spawned seeds.
2. Seed animation.
3. Seed filtering by visibility testing and rejection.
4. Curve extrusion from seeds.
5. Triangle strip extrusion from curves.
6. Stroke weighting and rendering.

Along the process, various weighting factors are computed for the seeds—w^{prox} for proximity to isosurface, w^{age} for age , w^{vis} for visibility, w^{rej} for density control by rejection. The product of these w^{Π} is used in the final rendering step for opacity weighting, with the optimization that seeds with zero weight need not be extruded into hatch strokes. The weight without density control, $w^{\text{pre}} = w^{\text{prox}} w^{\text{age}} w^{\text{vis}}$ is used for estimating *pre-rejection density*.

When seeds are initialized, they are placed randomly on atom-centered spheres within the effective radius. They are not guaranteed to be on the compound isosurface, and the isosurface-projected distribution might not be uniform. Those requirements are to be achieved by consequent seed animation and rejection steps, over the course of several frames. Seed points are re-initialized after a fixed lifetime to avoid excessive clustering. Seed point ages are evenly distributed, so that only a small percentage of seeds are re-initialized in every frame. Weight w^{prox} is computed as a smooth step function on the difference of the field value at the seed point and the desired isosurface. This is to eliminate seeds not yet converged to the surface. Weight w^{age} fades to zero at the beginning and the end of the seed lifetime to avoid suddenly appearing and disappearing hatch lines.

Seed point animation is based on the technique proposed by [21], without using repulsion forces to achieve

uniform density, thus eliminating the need for a self-spatial join on seeds. Seed animation according to Section 4.3 requires the field value and the gradient. We compute these, and also the world space *stroke direction* along the isosurface. The computation of the stroke direction involves first finding the pure and mixed second derivatives forming the Hessian, the principal curvatures and curvature directions, the determinant D indicating whether the seed is at an umbilical point, the cosine of the view angle $\cos\Theta$ indicating whether the seed is near a silhouette, and the local illumination V at the seed, normalized to a desired tone.

Generally, the stroke direction is the principal curvature direction of the isosurface, but near umbilical points, we employ a custom direction, obtained as the cross product of the surface normal and a per-atom direction vector. The choice of this per-atom vector might be random, or subject to artistic consideration. In order to produce simple outlines, a different direction scheme is applied to lines near the silhouette: the stroke direction there is perpendicular to both the view direction and the surface gradient (see Figure 2). The three direction schemes are combined based on D and $\cos\Theta$, so that there are no abrupt changes in the stroke direction. For any direction $\mathbf{t}$, the corresponding curvature κ can be found as $\kappa = \kappa_1 (\mathbf{t}\cdot\mathbf{t}_1)^2 + \kappa_2 (\mathbf{t}\cdot\mathbf{t}_2)^2$.

Seed points have to pass two filters to see if they should be extruded into hatch strokes. The first is the visibility test needed to decide if the seeds are seen from the camera. For this purpose, we render all seeds as isosurface-oriented billboards into a low-resolution buffer, outputting fragment depths and their squares. The purpose here is to approximate the depth of the isosuface itself by using the depth values of the billboards covering it. Using the idea of *variance shadow maps* [4], this low-resolution depth map is heavily filtered by two-pass separable Gaussian filtering. The resulting approximate variance depth map can be used for a smooth and lenient rejection of hidden seeds, producing visibility factor w^{vis}. Using this visibility factor to modify hatch stroke opacity causes strokes at and behind the boundaries of the surface to fade out smoothly, enabling partially visible strokes to appear. As we are emulating the hand-drawn style, the error—from approximating the isosurface with billboards, using a low-resolution map, aggressive filtering, and testing for visibility only at seeds—is not only acceptable, but welcome.

The second rejection step is to achieve an illumination-dictated screen space density of seed points (Figure 3). The *full cover density* Υ^{full} is an artistic parameter that specifies the seed density corresponding to surfaces devoid of illumination. This, modulated by seed tone V_k gives the desired on-screen density near a seed. Let us refer to the local density of all screen-projected seed points (weighted by w^{pre}) as Υ^{pre}. The ratio of $V_k\Upsilon^{\text{full}}/\Upsilon^{\text{pre}}$ gives the percentage of seed points to be kept. If all seed points have a random normalized priority value p_k, then those with priorities above the desired percentage should be rejected. The Υ^{pre} density is approximated by rendering all visible seeds, extruded into approximate hatch strokes, with additive blending, weighted by w^{pre} into a low-resolution buffer, and performing heavy filtering to eliminate rasterization artifacts. Note that what we get is not exactly the density of seeds, but an approximate density of hatching coverage. Thus, it helps to eliminate not only the clustering of seeds, but also the clustering of aligned strokes. Weight w^{rej} is computed as a smooth step function of $V_k\Upsilon^{\text{full}}/\Upsilon^{\text{pre}} - p_k$. Thus, rejection is performed smoothly, thus avoiding temporal visual artifacts, i.e. suddenly disappearing, appearing, or flickering hatch lines.

The seeds surviving visibility testing and rejection are extruded into curves. For short strokes, it is sufficient to use the local curvature at the seed, but longer lines require integration along the isosurface. In the latter case, visibility testing has to be performed for all samples. Curves are extruded into triangle strips to a uniform image space width. This width, and also the length of strokes, is an artistic parameter.

In the final rendering step the stroke is textured with an artist-drawn stroke image, with weights applied as opacity modifiers. We only discard the seeds if the weight would indeed be zero.

6 Implementation

The steps of our algorithm are implemented in five passes, depicted in Figure 4.

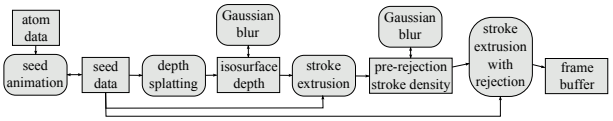


Figure 4: Shader passes of the implementation.

The first pass performs seed animation. All seed data is stored in textures used as data tables, where rows correspond to atoms, and the elements of the rows are individual seed points. Aging and re-initialization of seeds is performed by a rotating pipeline. In fact, in every texture row, seed attributes are shifted out to the right and reinitialized seeds shift in from the left, at a constant rate. The textures are also shifted vertically, to account for newborn and dying atoms, if so dictated by fluid simulation. For computation of quantities derived from the field function we used a regular grid space subdivision scheme to access relevant atoms.

The second pass produces the variance depth map of the isosurface to be used for a visibility filtering. Billboards are only extruded for seeds already converged to the surface to avoid unnecessary occlusion by seeds that are still trying to find their place. The depth values are blurred using a Gaussian filter, in accordance with the VSM technique, eliminating jagged edges in the depth map that could cause flickering hatch strokes in the final image.

The fourth pass is used to produce an image of Υ^{pre} values. These are needed for rejection of seeds later, to

Figure 2: Hatching of an LSD molecule discarding seeds near silhouettes (left) and rotating strokes to produce outlines (right).

achieve uniform screen space density. It extrudes hatch strokes from all visible seed points, and applies the same opacity weighting to them—for visibility, age and proximity to the isosurface—, as would be when rendering on-screen strokes. Rejection for density, however, is not applied, since the goal is to approximate the hatching density from all visible seeds. After visibility determination and curve extrusion, the hatch strokes are rendered, given color and opacity values that smoothly fall off towards the edges of the strokes. The output of this pass is rendered to a texture, using additive blending to generate high density values for high density areas on the screen.The Υ^{pre} density values also need to be blurred, to avoid rasterization artifacts caused by jagged edges of approximate hatch strokes.

In the final pass, the process of rejection and opacity weighting based on visibility and hatch stroke extrusion is the same as it was during rendering the Υ^{pre} density. In addition, this pass also weights seed points using the Υ^{pre} values, and illumination values calculated on the fly, before extruding the hatch strokes themselves. If the compound weight of the seed is positive, the strokes are extruded, textured, and opacity is modulated by all weighting factors.

7 Results and future work

We ran our tests on a PC with an ATI5850 graphics card. At a resolution of 1024×768, with 65K seeds, which we deemed sufficient for rendering quality, and regardless of the number of atoms, we measured frame rates around 20 FPS.

Extruding long hatch curves requires several curvature samples on the isosurface, and as curves travel into zones of different curvature characteristics they tend to cross each other. Density estimation at seeds is also less accurate in this case. Therefore, we wish to investigate the possibility of using several linked seeds points for every hatch curve. Another limitation of the method is that the seed density cannot exceed what is provided by rendering all seeds at unit weight. This is made worse if the distribution of seed points gets uneven because of seed motion. Thus, we plan to add seed fissioning and killing to improve performance and provide much wider level-of-detail support without increasing the seed count.

8 Acknowledgments

This work has been supported by OTKA PD-104710 and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

[1] Alexander G Belyaev, Alexander A Pasko, and Tosiyasu L Kunii. Ridges and ravines on implicit surfaces. In *Computer Graphics International, 1998. Proceedings*, pages 530–535. IEEE, 1998.

[2] J.F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics (TOG)*, 1(3):235–256, 1982.

[3] N.K.R. Bolla. High quality rendering of large point-based surfaces. Master's thesis, International Institute of Information Technology, Hyderabad-500 032, INDIA, 2010.

[4] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the 2006 symposium*

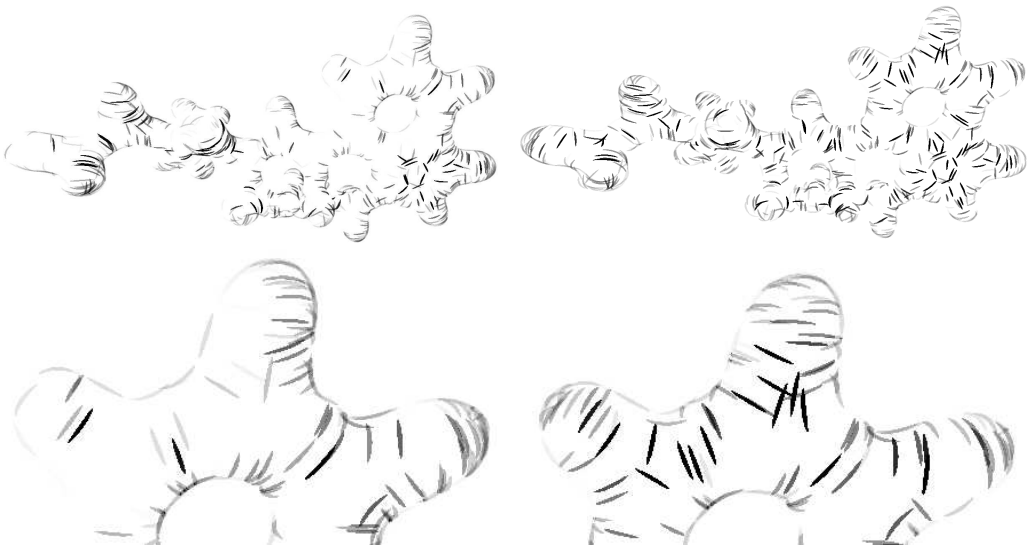


Figure 3: Uniform hatching of an LSD molecule with and without illumination.

on Interactive 3D graphics and games, pages 161–165. ACM, 2006.

[5] Gershon Elber. Interactive line art rendering of freeform surfaces. In *Computer Graphics Forum*, volume 18, pages 1–12. Wiley Online Library, 1999.

[6] Ahna Girshick, Victoria Interrante, Steven Haker, and Todd Lemoine. Line direction matters: an argument for the use of principal directions in 3d line drawings. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 43–52. ACM, 2000.

[7] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526. ACM Press/Addison-Wesley Publishing Co., 2000.

[8] Y. Kanamori, Z. Szego, and T. Nishita. GPU-based fast ray casting for a large number of metaballs. In *Computer Graphics Forum*, volume 27, pages 351–360, 2008.

[9] Yongjin Kim, Jingyi Yu, Xuan Yu, and Seungyong Lee. Line-art illustration of dynamic and specular surfaces. In *ACM Transactions on Graphics (TOG)*, volume 27, page 156. ACM, 2008.

[10] Yunjin Lee, Lee Markosian, Seungyong Lee, and John F Hughes. Line drawings via abstracted shading. In *ACM Transactions on Graphics (TOG)*, volume 26, page 18. ACM, 2007.

[11] Zoltán Lengyel, Tamás Umenhoffer, and László Szécsi. Screen space features for real-time hatching synthesis. In *Proceedings of the 9th conference of the Hungarian Association for Image Processing and Pattern Recognition*, KEPAF '13, pages 82–94, 2013.

[12] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *ACM Siggraph Computer Graphics*, volume 21, pages 163–169. ACM, 1987.

[13] Barbara J Meier. Painterly rendering for animation. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484. ACM, 1996.

[14] H. Nishimura, M. Hirai, T. Kawai, T. Kawata, I. Shirakawa, and K. Omura. Object modeling by distribution function and a method of image generation. *The Transactions of the Institute of Electronics and Communication Engineers of Japan*, 68(Part 4):718–725, 1985.

[15] T. Nishita and E. Nakamae. A method for displaying metaballs by using bézier clipping. In *Computer Graphics Forum*, volume 13, pages 271–280. Wiley Online Library, 1994.

[16] Afonso Paiva, Emilio Vital Brazil, Fabiano Petronetto, and Mario Costa Sousa. Fluid-based hatching for tone mapping in line illustrations. *The Visual Computer*, 25(5-7):519–527, 2009.

[17] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, page 581. ACM, 2001.

[18] László Szirmay-Kalos, György. Antal, and Ferenc Csonka. *Háromdimenziós grafika, animáció és játékfejlesztés*. ComputerBooks, Budapest, 2003.

[19] T. Umenhoffer, L. Szécsi, and L. Szirmay-Kalos. Hatching for motion picture production. In *Computer Graphics Forum*, volume 30, pages 533–542, 2011.

[20] W.J. van der Laan, S. Green, and M. Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, pages 91–98. ACM, 2009.

[21] K. van Kooten, G. van den Bergen, and A. Telea. Point-based visualization of metaballs on a GPU. *GPU Gems*, 3:123–148, 2007.

[22] Emilio Vital Brazil, Ives Macêdo, Mario Costa Sousa, Luiz Velho, and Luiz Henrique de Figueiredo. Shape and tone depiction for implicit surfaces. *Computers & Graphics*, 35(1):43–53, 2011.

[23] Georges Winkenbach and David H Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 91–100. ACM, 1994.

[24] Andrew P Witkin and Paul S Heckbert. Using particles to sample and control implicit surfaces. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 269–277. ACM, 1994.

[25] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The visual computer*, 2(4):227–234, 1986.

[26] Johannes Zander, Tobias Isenberg, Stefan Schlechtweg, and Thomas Strothotte. High quality hatching. In *Computer Graphics Forum*, volume 23, pages 421–430. Wiley Online Library, 2004.

Adaptive Tessellation in Screen Space Curved Reflections

Attila Szabo*
Supervised by: Reinhold Preiner

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Vienna / Austria

Abstract

Mirroring objects play an important role in the rendering of every-day scenes, as they aid in the recognition of materials, objects and the distance relations between them. Due to their complex nature, an accurate solution generally requires an expensive computation, which is mostly done using methods based on ray tracing. To reduce the workload, recent methods try to perform the computation in screen-space. However, in order to ensure accurate reflections, the geometry of the scene needs to be sufficiently tessellated to reduce the artifacts created by the linear interpolation of the GPU rasterizer. This creates a vast preprocessing effort and storage-overhead for the tessellated vertices.

In this paper, we present a method that performs this tessellation on the fly, reducing the error in the reflective image by inserting extra vertices where necessary. We prove the effectiveness of our approach in comparison to the state of the art and discuss limitations and ideas for possible future work.

Keywords: computer graphics, rendering, real-time rendering, deferred shading, tessellation, reflections

1 Introduction

In rendering, it is oftentimes the goal to render mirrors and reflective objects. Materials like metals and glass, make the scene feel believable to the viewer and contribute to the realism and beauty of images. Perfectly mirror-like surfaces can be often found in many man-made environments, such as in washroom appliances or cars.

In rendering systems, rendering specular reflections can be viewed as the process of finding *reflection points*, the places of reflection visible from the camera, and then reflecting the *reflected points* radiance at the reflection points toward the camera [10] (Figure 3(a)).

For rendering reflections, several methods and techniques have been developed. Usually, they either aim for maximizing *realism*, the physical accuracy, or *believability*, in which case the result will often only be "correct enough" for it to look relatively accurate to the viewer, but gain faster performance, be easier for a designer to work with, look aesthetically more pleasing, or similar.

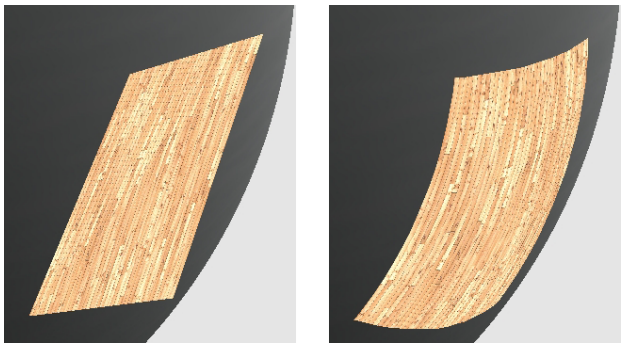


Figure 1: *Left:* Erroneous reflection of a square. Its sides are linear interpolations between the corners' reflections. *Right:* Correct reflection after sufficient tessellation of the square.

The efficient rendering of planar mirrors has been examined extensively [7]. However, the case of **curved** surface reflectors requires special consideration since the reflections can become exceedingly complex. Light rays are traced up to the reflectors, the reflected rays computed and recursively traced until a non-specular surface is reached. In general scenes, the number of light rays and recursive computation steps can become very high. Therefore, effective storage of scene geometry and specialized processing of light rays are needed to guarantee robust performance.

CPU solutions usually create and maintain sophisticated scene data structures and optimized calculations to achieve good performance, while most GPU based techniques take advantage of the fast GPU rasterization capabilities [10]. The GPU processor is usually given access to the scene geometry by storing it in uniform parameters or in textures [10].

Both approximate [1] and accurate [11] methods have been developed to tackle this problem, trading performance for precision and vice versa. Screen-space methods for reflection rendering [4] are especially attractive since they are able to maintain both good accuracy and performance. Here, *Deferred Shading* [3] is used to relay the computation of a reflected image to a second rendering

*e0925269@student.tuwien.ac.at

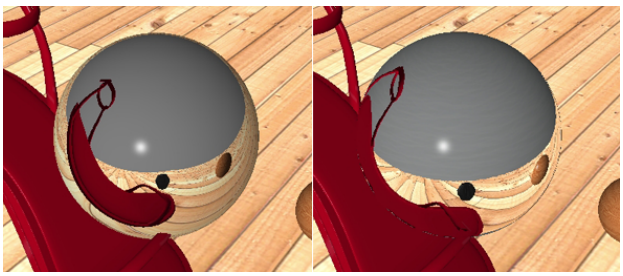


Figure 2: *Left:* Approximate reflection using Environment Mapping with errors. *Right:* Accurate reflection.

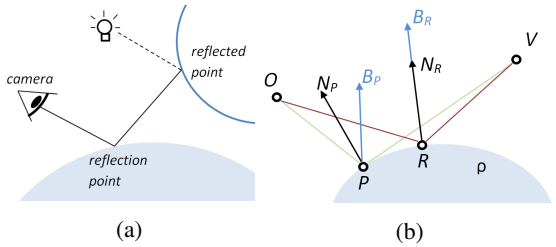


Figure 3: (a) Reflections are found by tracing light paths. (b) The relation between the viewpoint O, the world vertex V and a point P on the reflector surface ρ. The reflection point R is such that their bisector vector B_R and the surface normal N_R coincide.

pass. First, the camera perspective of each mirror's surface is rendered into a series of textures. Then, the reflection of an object is rendered on top of it by mapping all its vertices onto their reflection points on a mirror's screen space projection (the *virtual geometry*) and exploiting the graphics hardware to triangulate the full reflected image.

This method is generally fast, but can lead to artifacts. In the virtual geometry reflected by the mirror, triangles can become curved patches, and triangle edges can become curves. However, the GPU is built to rasterize only non-deformed triangles with straight edges. A denser tessellation of one triangle reduces the size of triangles allowing edges to be closer to the curvature of the mirror. For the reflection to be free of artifacts, the geometry of the scene needs to be sufficiently tessellated, as shown in Figure 1. However, pre-tessellating a whole scene is generally unfeasible, since storing and processing a high number of vertices can quickly get computationally too expensive.

In this paper, we introduce an extension to this method to improve the visual quality of the resulting image. Our approach tessellates the geometry on the fly and only in the parts where it makes a visible difference. We control this mechanism by using a simple and flexible error metric. We show that our approach can efficiently produce accurate reflection images on curved mirrors without any pre-tessellation of the scene, while maintaining a good rendering performance.

2 Related Work

Accurate reflections are commonly computed using *Ray Tracing* [11]. In this approach, for every pixel, a number of viewing rays are cast into the scene and their interactions calculated. The main disadvantage of Ray Tracing is its high computational cost, since it generally requires a very high number of viewing rays and the interactions may be complex. *Online* rendering usually implies heavy performance constraints in order to retain interactivity. While improvements have been proposed to make the algorithm work effectively on graphics hardware [9], Ray Tracing is still not always suitable to provide interactive frame rates in many cases and mostly relies on building and maintaining spatial data structures [12]. This is especially problematic in *dynamic scenes*, in which these data structures have to be rebuilt or updated when objects move.

Environment Mapping [2] allows an approximation of the reflection to be found very rapidly. In this approach, the environment around the reflector is rendered into a texture, such as a *cube map*. When shading a pixel belongig to a reflective surface, the reflected surface point in the environment is looked up in the cube map. However, this approach is not always physically accurate. Since environment mapping assumes that the environment is infinitely distant from the object, the reflection is approximately correct if the scene is sufficiently far away from the reflector surface [6, Chapter 7]. Figure 2 shows an example comparison between approximate and accurate reflections.

Reflections in planar mirrors are usually rendered by drawing the scene twice - once from the common viewpoint and once from the viewpoint reflected on the mirroring plane. The mirror image is then drawn on top of the mirror in the original image. *Non-Planar* mirrors however require a more sophisticated treatment, as their reflections cannot be modelled by linear projection as in the planar case above. The rendering technique proposed by Estalella et al. [4] addresses this circumstance. It follows the same idea of rendering virtual geometry inside a mirror, but extends it to curved mirrors. In the first rendering pass, the mirror's surface positions and normals are rendered into a series of textures. In the second pass, for each vertex in the scene, a pixel-by-pixel search across these textures is used to find the point that comes closest to its actual reflection on the mirror. To find this point, the following principle is used:

Consider a vertex V of world geometry that is to be reflected and the virtual camera's viewpoint O. For every point P on the surface of a curved reflector ρ the *bisector vector* B_P of the angle between O and V in P can be defined, as well as the curved reflector's *surface normal* N_P, see Figure 3(b). The *point of reflection* R on ρ is such that its bisector vector B_R and its surface normal N_R coincide. This point of reflection is unique across closed convex reflectors [5]. We use this principle later in Section 4.2 to determine adaptively the required degree of tessellation.

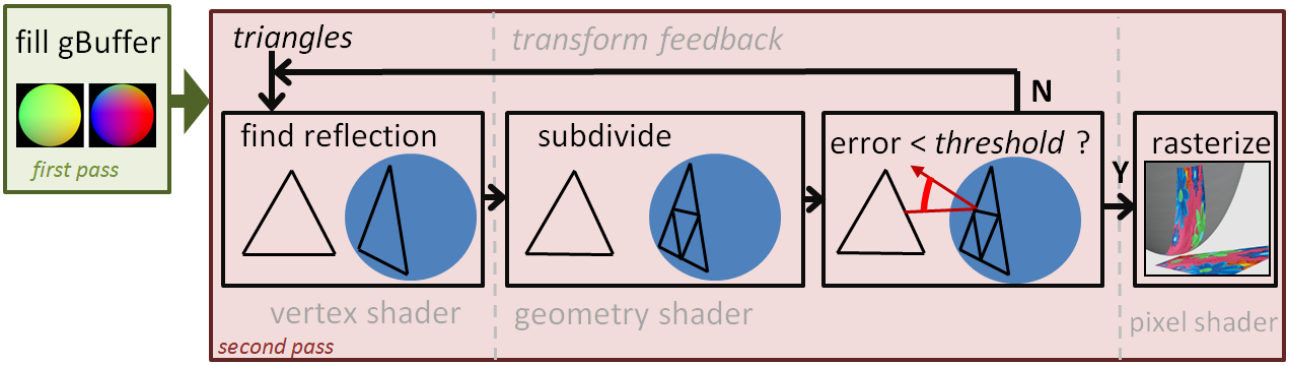


Figure 4: Overview of the rendering passes.

The result is a set of virtual "mirrored" vertices, which are triangulated and rasterized on top of the original image. Although the mirror image is accurate for each of the vertices, their triangulation only represent a linear approximation to the correct non-linear curved mapping. Therefore, geometry must be tessellated fine enough for a sufficient piecewise linear approximation in screen space.

3 Overview

Reflections are rendered in a multi-pass approach. An overview is given in Figure 4. In the first pass, the mirror's surface positions and normals are rendered into the *gBuffer*. In the second pass, the gBuffer is used to calculate the reflection points of all vertices in the scene in the *vertex shader* stage (Section 4.1). The vertices are stored as vertex-triples forming triangles. The triangles are passed on to the *geometry shader* stage where based on a *reflection error* metric we check for each triangle whether whether it is sufficiently tesselated. If so, they are finalized for rasterizing. If not, the triangles are subdivided into four equal triangles and fed back to the vertex shader stage using *transform feedback*. The evaluation and subdivision of triangles is described in Section 4.2. The iterative subdivision of triangles continues until all triangles are finalized or an iteration limit has been reached. Finally, the triangles are rasterized to render the reflection.

4 Implementation

4.1 Screen Space Reflection

The system assumes the scene to consist of *triangle* primitives, which are marked to be *mirrors* or *non-mirrors*. Each scene object's vertices have a world-space position and a surface normal vector of the surface they belong to. The procedure of rendering accurate screen-space reflections for the mirroring scene is outlined by Algorithm 1.

Each reflector is rasterized from the camera's point of view, and its world-space position and surface normals stored in two 2D textures (*Position Map* and *Normal Map*).

```
foreach NonMirror n do
    Draw(n);
end
foreach Mirror m do
    gBuffer ← RenderGbuffer(m);
    foreach NonMirror n do
        DrawReflection(gBuffer,n);
    end
end
```

Algorithm 1: Functional outline of how a frame is rendered.

```
function FindReflectionPoint(gBuffer,
vertex)
CurrentPixel ← MirrorCenter();
repeat
    PreviousPixel ← CurrentPixel;
    CurrentPixel ← BestNeighbour();
until PreviousPixel == CurrentPixel;
return GetPosition(gBuffer ,CurrentPixel);

function GetPixelError(gBuffer, Pixel)
S ← GetPosition(gBuffer,Pixel);
N ← GetNormal(gBuffer,Pixel);
po ← normalize( CameraWorldPosition - S);
pv ← normalize( VertexWorldPosition - S);
bisector ← normalize( po +pv);
return dot( bisector, N);
```

Algorithm 2: The method for calculating the reflection error of one pixel.

These two maps together are referred to as `gBuffer` [8, Chapter 9] in the following.

The next processing step is executed inside a vertex shader. The input of the vertex shader are individual vertices. The resulting vertices are passed on to the geometry shader. There triples of vertices are interpreted as triangles to be either drawn directly or subdivided, the resulting vertices being fed back into this stage.

Given a mirror's `gBuffer` and a vertex, Algorithm 2 shows how to find the vertex' reflection point on the mirror surface. The function starts its search at the the center of the reflector in screen space, and then iteratively steps towards the pixel position of the reflection point. At each currently considered pixel, its four directly neighbouring texels are examined and their reflection error calculated. This error is computed in the `GetPixelError` function.S and N are the position and the normal of the mirror surface point stored in this pixel, and po and pv are the direction vectors from the surface point S to the camera and the vertex position, respectively (see Figure 3(b)).

The deviation of the current pixel location to the sought reflection point (i.e., the *reflection error*) is measured by the dot product between the bisector vector between these two direction vectors and the reflector surface normal. This error is calculated for all four neighbours and the current pixel and then simply considers the neighbouring pixel with the lowest reflection error (highest dot product). This process is repeated until no neighbouring pixel with a lower reflection error current one can be found, at which point the final reflection point is found.

To ensure correct visibility when rendering a vertex, the z-buffer needs to be updated according to the reflected depth, i.e., the distance between the vertex and its reflection.

Some vertices do not have their reflection point on the visible surface of the reflector. For a reflector with closed uniformly convex geometry, such as a sphere, these are the vertices hidden behind the projection of the reflector in screen space. Taking such hidden reflection points into account for triangulation would result in incorrect triangles, and therefore have to be discarded. We identify such hidden vertices using the condition [4]:

$$pv \cdot N < 0$$

Vertices for which this condition is true are marked and their corresponding triangles are discarded. In addition, for reflectors which are not closed, such as reflectors that are partially obscured, the reflection point is found when the search terminates at the edge of the reflector projection [4].

4.2 Adaptive Tessellation

To address the problem of reflection artifacts for low-poly geometry (Figure 1), we perform an adaptive tessellation at render time.

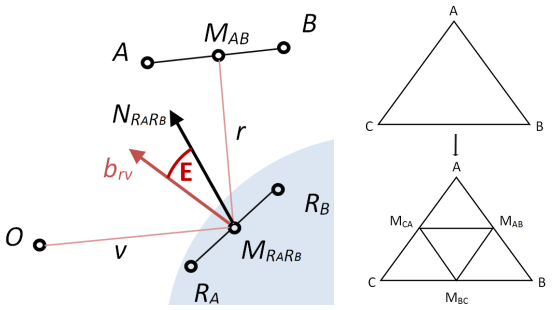


Figure 5: (a) The reflection error E. (b) The subdivision rule used to tessellate a triangle.

This rendering stage is implemented in the geometry shader. The input are triples of vertices, forming *triangles*, resulting from the vertex shader stage (Section 4.1). The triangles are evaluated for subsequent subdivision. The resulting triangles are written to one of two vertex buffers, the *working buffer* and the *finished buffer*, using transform feedback. This rendering stage is outlined in Algorithm 3.

```
function adaptiveTess(A, B, C)
E1 ← calcError(A,B) ;
E2 ← calcError(B,C) ;
E3 ← calcError(A,C) ;
triError ← max(E1, E2, E3) ;
if triError < threshold then
    streamOut( finished);
else
    subdivide();
    streamOut( working);
end
```

Algorithm 3: Overview of adaptive tessellation function.

An input triangle is defined by its three vertices A, B and C, whose reflection points were calculated in the vertex shader stage. To decide whether a triangle is to be tessellated, the *reflection error* `triError` of the triangle is defined. The edge errors E are calculated for each of the three *edges*, formed by pairs of vertices, and `triError` is the maximum of the three E. For one edge, E is calculated in the function `calcError` using the following formula:

$$E = 1 - \frac{N_{R_A R_B} \cdot b_{vr} + 1}{2}$$

This relation is visualized in Figure 5(a). $N_{R_A R_B}$ is the normal at the linearly interpolated median point between R_A and R_B on the reflector surface, v is the viewing ray direction from the median point to the camera position, r is the direction from the median point to the median between the original world vertices A and B, and b_{vr} is the normalized bisector between the two. The dot product is

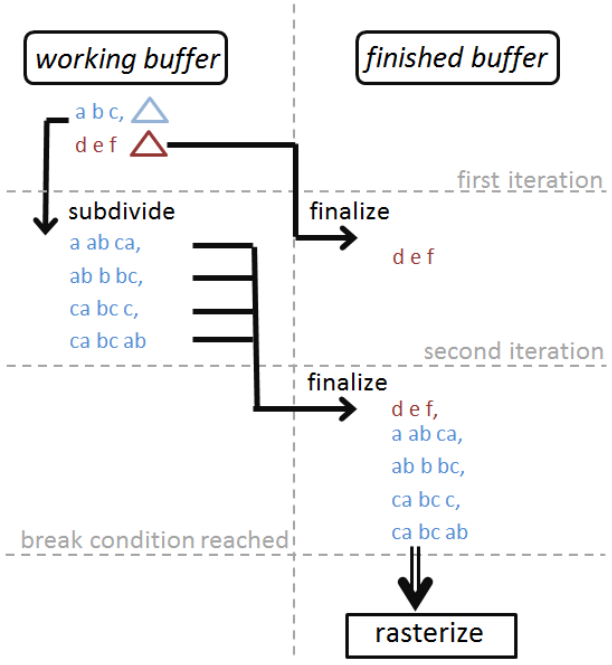


Figure 6: The tessellation loop iterates until the working buffer is empty or maximum iterations is reached.

normalized to lie inside $(0, 1)$, where 0 means no error. E describes the difference between the linearly interpolated reflection of an edge and the one showing accurate curvature.

`triError` is compared against a user-set *threshold*. The threshold describes the highest acceptable deviation from the accurately curved reflection. If `triError` is less than the threshold, the triangle is deemed sufficiently tessellated and is *finalized*. Its three vertices are appended to the finished buffer using transform feedback (function `streamOut`).

If `triError` is greater than the threshold, the triangle is `subdivided` (function `subdivide`). The subdivision rule used is shown in Figure 5(b). The three halfway points along the edges M_{AB}, M_{BC} and M_{CA} are used as vertices with the three triangle vertices to form four equal triangles. The twelve vertices forming the four new triangles are written into the working buffer (function `streamOut`).

The rendering pass is repeated using the working buffer as input for the vertex shader stage. After their vertices being reflected, the triangles reach the geometry shader stage again to be finalized or further subdivided. This *tessellation loop* iteratively refines the tessellation of triangles until either no vertices are written into the working buffer, or a maximum number of iterations (the *tessellation level*) is reached. A visualization of the loop can be seen in Figure 6. If the iteration limit is reached, remaining vertices in the working buffer are copied into the finished buffer.

The tessellation level ensures that there is a hard limit to how often a triangle can be subdivided. The subdivision limit is usually not reached, unless the error threshold is set very low (close to 0), in which case subpixel accuracy is reached and further subdivisions can be limited. In addition, the limit avoids an infinite loop when the threshold is equal to 0.

After this rendering pass, the vertices in the finished buffer are rasterized. The result is a rendering of an accurate mirror image.

5 Results

5.1 Rendering Quality

Adaptive tessellation allows to render scenes as if they were fully tessellated. Figure 7 shows a scene being reflected in a mirroring ball. The scene contains both models with a very coarse and a very high original degree of geometry tessellation. Without adaptive tessellation (left), the result is visibly wrong. The tablecloth and candlestick are modelled using only a small number of quads. The corners of those quads are reflected correctly, but the linear interpolation between them does not suffice for a correct mirror image. The teapot is modelled with many more vertices and therefore produces a reflection image of much better quality. A full tessellation of the entire scene using four subdivision iterations is shown at the right image. For the teapot, this adds a lot of superfluous vertices, since the reflection does not improve. Using adaptive tessellation (middle) allows us to address both these problems. Coarse models are tessellated until quality of the rasterized reflection image is sufficient. On the other hand, geometry with already sufficient degree of detail, are not further subdivided, when drawing their reflection image.

Adaptive tessellation is very robust regarding different circumstances. Both simple reflections and complex surfaces are handled as accurately as needed. Our proposed error metric is derived from the screen space accuracy of the reflection and it relates directly to the errors visible in the rendered image. The error threshold represents a tradeoff parameter between quality and performance. It can be adjusted, even during runtime, to accommodate either faster performance or more accurate images. In fact, one could set the threshold to a sufficiently small value (subpixel size) to eliminate all visible artifacts.

As shown in Figure 7, our method can produce accurate reflections without the need for any pre-tessellation of the scene. It follows that content creators need not worry about specifically adjusting their models, and the technique can be implemented in a rendering system without big impact on established functionality.

5.2 Performance

The error threshold parameter allows for trading between accuracy and performance.

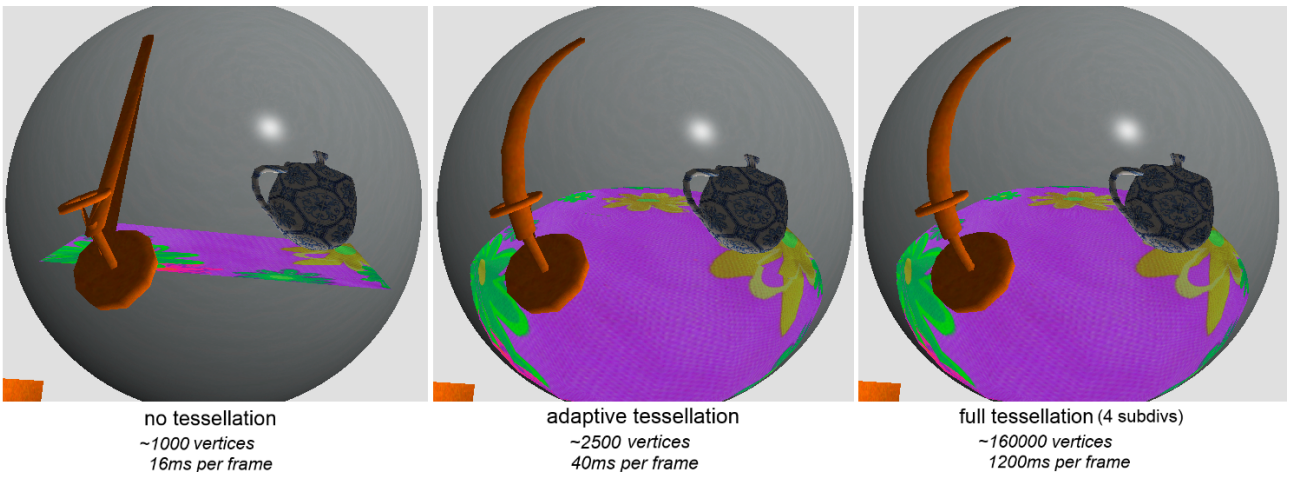


Figure 7: An example scene containing both coarse and fine meshes.

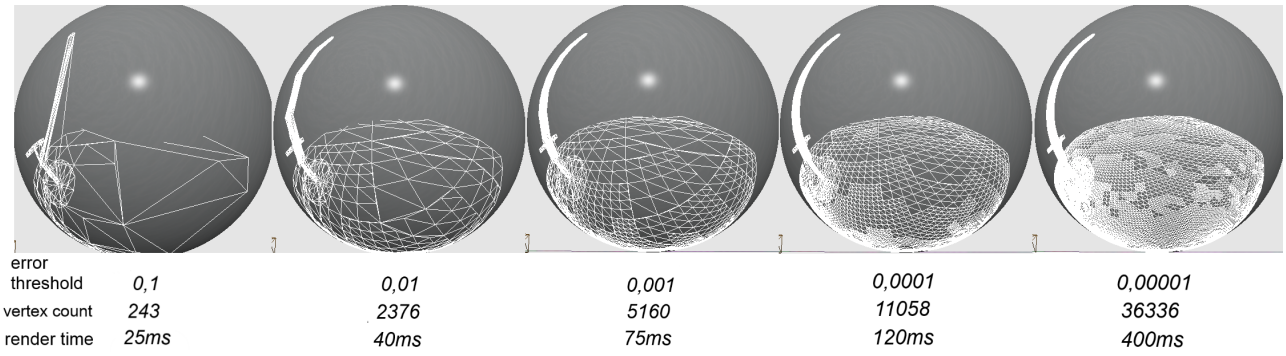


Figure 8: Adaptive tessellation with different error threshold values.

Figure 8 shows the results of using different thresholds. The rendering has been performed on a PC with an *Intel Core2Duo* CPU and a *Nvidia GeForce GTX 260* graphics card. If the threshold is higher, larger errors are allowed and fewer subdivisions are performed. A too relaxed threshold results in a reduction of quality and introduces artifacts. We found that a value of 0.1 or larger is too high. A value between 0.1 and 0.01 generally creates perfectly acceptable results while maintaining goog performance. In particular, high curvature surface parts are subdivided often enough to provide accurate results. If the threshold is set even smaller, close to 0, the geometry is tessellated very finely. In this case, interactive performance can not be provided anymore. The value 0 itself causes *full tessellation* to be performed, in which case all triangle subdivision is repeated until the tessellation level is reached. Conversely, 1 stands for *no tessellation*.

Figure 9 shows a comparison of resulting vertex numbers between full, adaptive and no tessellation in the example scene (Figure 7). It can be seen that adaptive tessellation results in fewer vertices compared to full tessellation, requiring fewer expensive reflection point searches.

Furthermore, the performance of adaptive tessellation does not depend on any spatial data structures that cause a maintenance overhead. Therefore, dynamic scenes, in which objects move or are otherwise animated, are handled without negative impact on the performance.

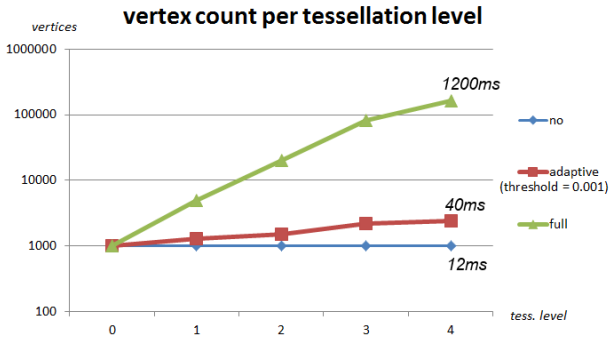
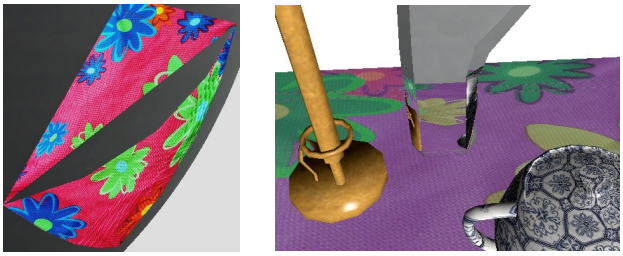


Figure 9: Number of vertices resulting from different tessellation levels in the scene from Figure 7.

5.3 Limitations

As shown in Figure 10(a), our subdivision pattern can result in holes appearing in the reflection where triangles of different tessellations meet. The size of the hole relates to the difference in error of the two neighbouring triangles.



(a) (b)

Figure 10: (a) A hole between triangles with few and many subdivisions. (b) A simple polygonal mirror.

However, the error is only this severe if the camera is at a steep angle and a close distance to the reflector. In addition, with our error threshold the extent of the gaps is easily controllable. If the use of the threshold is restricted, one could instead imagine an extension in which the threshold is adjusted dynamically based on the probability of such holes appearing.

In addition, the base algorithm we use for finding reflection points assumes that vertices only have one reflection point [4]. As mentioned above, this holds for all convex mirror surfaces, and for concave surfaces of sufficiently large distances to the reflected geometry. Figure 10(b) shows an example for an arbitrary polyongal mirror.

5.4 Future Work

The problem of holes appearing between triangles could be solved by using different patterns of triangle subdivision. We use our subdivision rule because it is computationally simple and equally suited for any kind of reflector surface. However, it could be investigated to use irregular subdivision of triangles to achieve the same level of tessellation along shared edges, which would prevent holes from appearing between them.

Another improvement to the algorithm would be to extend it to arbitrarily shaped mirrors. This is a property of the underlying reflection point search algorithm. It could be solved by finding a method to split up the mirror into segments of uniform curvature and finding the reflection on each of them [4].

6 Conclusion

In this paper a method for rendering an accurate reflection on the surface of a curved reflector in real-time has been examined. It is a multi-pass approach in which first the image space reflection point of each vertex is found. The triangles formed by the vertices are tessellated adaptively according to an error metric, which is based on the difference in quality resulting from a subdivision iteration. Subdivision steps are skipped if they do not cause a noticeable effect in the final image, greatly reducing the number of vertices needing to be reflected. The subdivision is repeated until the triangles are sufficiently tessellated. Finally, the reflected geometry is rasterized by the graphics hardware. The method can provide interactive framerates for dynamic scenes. Discussed results show that the technique examined in this paper is a robust choice in real-time rendering and may well serve as an anchor point for future considerations extending its applicability.

References

[1] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, August 1978.

[2] James F Blinn and Martin E Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.

[3] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 21–30. ACM, 1988.

[4] Pau Estalella, Ignacio Martin, George Drettakis, and Dani Tost. A GPU-driven algorithm for accurate interactive reflections on curved objects. In *Proceedings of the 17th Eurographics conference on Rendering Techniques*, EGSR'06, pages 313–318, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[5] Pau Estalella, Ignacio Martin, George Drettakis, Dani Tost, Olivier Devillers, Frederic Cazals, et al. Accurate interactive specular reflections on curved objects. In *Vision Modeling and Visualization (VMV 2005)*, 2005.

[6] Randima Fernando and Mark J Kilgard. *The Cg Tutorial: The definitive guide to programmable real-time graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[7] Mark J Kilgard. Improving shadows and reflections via the stencil buffer. *Advanced OpenGL Game Development*, pages 204–253, 1999.

[8] Matt Pharr and Randima Fernando. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[9] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 703–712, New York, NY, USA, 2002. ACM.

[10] Lszl Szirmay-Kalos, Tams Umenhoffer, Gustavo Patow, Lszl Szcsi, and Mateu Sbert. Specular effects on the gpu: State of the art. *Computer Graphics Forum*, 28(6):1586–1617, 2009.

[11] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[12] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 126:1–126:11, New York, NY, USA, 2008. ACM.

Automated Lighting Design For Photorealistic Rendering

Silvana Podaras*
Supervised by: Károly Zsolnai†

Institute of Computer Graphics
Vienna University of Technology
Vienna / Austria

Abstract

We present a novel technique to minimize the number of light sources in a virtual 3D scene without introducing significant perceptible changes to it. The implementation is done as an extension of *LuxRender*, a state-of-the-art, physically based and open-source renderer. The algorithm adjusts the intensities of the light sources in a way that a set of light sources can be substituted by a smaller set, thus enabling to render a similar image with significantly less number of light sources, introducing a remarkable reduction to the execution time of scenes where many light sources are used.

Keywords: radiosity, global illumination, constant time

1 Introduction

With the advance of technology, the use of computer graphics has become an everyday routine in motion picture production. Since the making of "Toy Story", the first feature-length film that was entirely computer-animated, there have been a lot of improvements in technology. Nowadays, it is possible to create images that are almost indistinguishable from reality. One of the key elements in order to make a film look "real" is to simulate physically correct light transport when computing an image.

One of first attempts for such a simulation was made by Appel [1], who introduced the ray tracing algorithm. Although the images rendered with that method were far from photorealistic, the fundamental concept has become the basis of state-of-the-art algorithms. A major problem of photorealistic rendering is the time needed to gain high-quality images, because rigorous mathematical and statistical methods are used to simulate realistic effects. Depending on the desired effects and used hardware, render times can be prohibitively long - it can take up to hours or days to obtain images with satisfying quality [10].

Speeding up the rendering process has thus been a hot topic in science and industry for years, and was usually involving the creation of more efficient sampling strategies or better rendering algorithms. The problem can also be addressed from a different angle to speed up this process by means of reducing the number of light sources used in a scene. This idea came up when considering the way how industry giants like PIXAR use physically based ray-tracing systems in their daily work [4]. In order to achieve not only a physically plausible image, but also adhere to a certain look and feel desired by the artist, many light sources are placed in a scene - even up to hundreds of them [2]. Such vast number of light sources directly affects render time, because more of them have to be sampled in order to get a smooth and converged output. When having so many light sources in a scene, would it be possible to render an (almost) identical image with fewer light sources? For a large number of sources, the answer to this question can hardly be given in a reasonable amount of time when letting a user try out all different settings. Instead, the idea came up to let an algorithm perform the adjustment of the light intensities in order to find a similar result image which uses less lights than the initial setup. The overall concept how such an algorithm could be designed is explained in more detail in Section 3.

This work is founded on the assumption that using less light sources results in faster execution times. In order to show the validity of this assumption, an empirical evaluation on several scenes was done. Those measurements were made by means of testing on five carefully chosen scenes with a varying number of light sources, light sampling strategies and a comparison to the ground truth image. The results are presented in Section 5.

2 Related work

Although various efforts were made to speed up the rendering process, little research has been made to reduce the number of lights used in a scene. The most noticeable work is 'Lightcuts' by Walter et al. [13]. In this approach, lights in a scene are first clustered by spatial proximity and similar orientation. Those clusters get hierarchically organized in form of a binary tree. Every cluster is represented by one of the lights it consists of, and can be further refined to smaller clusters, which also use one light as representative, and so on. To reduce light sources, a "cut"

*spodaras@cg.tuwien.ac.at
†zsolnai@cg.tuwien.ac.at

through the tree is made and only the representative lights of the clusters in the cut are used to illuminate the scene. The representative light in a cluster gets modified in order to approximate the resulting illumination if all the lights in the cluster would be used. If the error of such an approximation is so small that it is not perceivable, only the representative light is used when rendering the scene, else the cluster is refined and more lights are used.

Apart from that approach, little prior work has been done to automatically reduce the number of light sources in a scene. Instead, the more general field of automated lighting design has been a popular topic. This field focuses on computationally adjusting parameters like for example light intensities and emission colors, instead of letting the user handle the fine-tuning.

The majority of methods let the user define a "desired" image as input, and the computer tries to calculate the according parameters to achieve this effect. Schoeneman et al. [11] for example assume that the designers of a virtual scene know where to place light sources, but fail in choosing the right intensities or colors. After having the designer "paint" effects such as shadows and spots of light on a target image, an optimization process is started to determine the settings that match the painted image best. Similar work has been done by Costa et al. [3] and Kawai et al. [6].

A contrary approach is "interactive evolution", which lets the user explore a set of possible solutions that the computer creates. Sims [12] used evolutionary algorithms in combination with user input to generate different sets of plant structures, procedural textures and animations. The quality for each solution is determined by the subjective judgment of the user, before the next evolution step is applied. Thus the user can "guide" the results in a specific direction without having to know about the underlying mechanisms for calculating the parameters.
The design galleries approach of Marks et al. [8] lets the computer set up many different light settings and present them to the user, who can choose the setup that seems most appealing to him.

3 Concept

When rendering a scene which makes use of many lights, eventually the same result image could be achieved by using fewer lights when turning some of them off or when changing their intensities. Trying out all variations manually is not an option, because the vast amount of possible settings would make this a time-consuming and cumbersome task. On the other hand, automatizing this process is relatively simple. An algorithm for this task would have to try out different light settings for a scene and compare the resulting images to an initial image the user wants to achieve. The more similar a new image is to the desired one, and the less light sources are used, the better the solution is.

Despite the idea itself is very simple, it is crucial to understand that the problem space is remarkably high dimensional. When trying to reduce the number of lights by only turning them on and off, a binary integer programming problem has to be solved, which is known to be in the NP-hard complexity class. Although the number of solutions increases with the number of lights used, the total amount is finite (up to $2^{amount\ of\ lights}$). This method would be sufficient for simple scenarios, where several light sources clearly have no contribution to the scene. This could be the case if an artist has created a light source with almost no intensity in the scene, or when a light source gets occluded by some object and suddenly has no contribution to the image. To decide if a proposed solution is close enough to the original image, a user-defined threshold could be set.

In practical applications, such easy scenarios will be the minority of cases. To reduce the overall amount of lights, changing the intensity of some of them before turning others off will be necessary. The solution space in this case varies drastically from the first one: an infinite amount of solutions exists, and each one is a valid image gained with certain light settings. For an arbitrary scene, it is unclear which light intensities have to be changed in what way. Exploring this search space without making more constraining assumptions is a non-trivial task. This problem can be addressed by using classical constrained optimization algorithms. In our work, we have used a genetic algorithm due to its capability to explore such vast search spaces. How such an algorithm was designed for our problem is described in the following paragraphs.

Genetic Algorithms (from now on referred to as "GA") are search heuristics inspired by the processes and mechanisms of natural evolution and can be applied to a large class of practical problems. The only requirement is that solutions for the problem can be encoded in a form that they can be processed by genetic operators such as mutation, crossover and fitness. In a GA, those encoded solutions are also called "chromosomes" [9], [5]. In our case, a possible solution can be represented by a vector of values which stores the intensities of each light. Those values are binary if the lights should simply be turned off and on, or they can encode continuous intensity values if the optimization routine should change the light intensities as well. A collection of initial solutions is generated by assigning the initial light settings to several chromosomes. Then, to gain new solutions, each chromosome gets randomly modified by either mutation or crossover operations. Mutation chooses one index of the vector at random and either flips the bit in binary-mode or adds or subtracts a small value for continuous optimization. The crossover operator simply chooses two chromosomes at random and recombines them at a randomly chosen index to form a new chromosome.

After the initialization, an evaluation step has to take place which determines how good a solution is (in terms of a GA: the *fitness* of a solution has to be evaluated). First, the amount of lights used would have to be calcu-

lated. This is done by summing up all non-zero components of a vector containing the individual light source intensities, i.e. $\vec{X_n} = [a_1, a_2, a_3, ..., a_n]$. This is done either by calculating the l_0 quasi-norm or l_1-norm of a vector:

$$\|X_n\|_0 = \sqrt[0]{\sum_{i=1}^{n} |a_i|^0}, \quad \forall a_i, a_i \in \mathbb{R}, \tag{1}$$

$$\|X_n\|_1 = \sum_{i=1}^{n} |a_i|. \tag{2}$$

The first case would be an integer optimization problem where lights only get turned off or on, the second metric adapts the lights intensities. However, using only the l_0- or l_1-norm of a vector to determine the fitness of a solution would be not sufficient, because it lacks the constraint that the output should be faithful to the input image.

A simple method to determine if a solution is good enough would be to define a threshold, which determines the maximum allowed difference between the initial image and a possible solution image. Calculating the difference is done by subtracting the images pixel-wise from each other and summing up the squared absolute differences. The solution is only considered as valid if the difference lies below the threshold, otherwise it is discarded. For valid solutions, the fitness gets calculated as the weight of a vector.
This trivial formalization is, however clearly not feasible as it leaves two problems unresolved. The first problem is that when creating an invalid solution, the algorithm never gets any kind of feedback on how close it was to a valid one. Not having this knowledge of how "close" a solution is makes optimization no better than any exhaustive sampling technique. The algorithm needs more elaborate feedback in order to know if the optimization is heading in a good direction. The second problem can be depicted by the following scenario: two valid solutions are available, both of them use three lights out of many. The three lights used in the first solution are different from the ones in the second - which of the two solutions is the better one, when both of them have the same fitness?

To solve both problems at once, the difference between the initial image and a temporary solution image is added after calculating the norm - in this way, the faithfulness of the output image is also considered, and with this knowledge, the algorithm can generate better and better solutions in every iteration. The overall fitness $f()$ of a solution vector $\vec{X_n}$ gets calculated as described in equation 3. The first term is the calculation of the norm $\|X_n\|_s$, as explained above in equations 1 and 2. The second term calculates the distance between the two images. This is done by summing up the squared differences of the pixels, where T_{ij} stands for the pixels of "target image" and C_{ij} for the pixels of the "current solution image". There are also two parameters for weighting the lights (p_L) and the difference (p_D). They act as a kind of quality switch and allow more control on the optimization procedure. The contribution to the weight of the vector of each light is multiplied by the factor p_L, therefore better fitness values are achieved if the algorithm tries to turn off lights instead of finding a solution which has little difference to the initial image. If on the other hand, a result image that's very close to the original is desired, the weight for the difference would have to be set to an appropriate value.

$$f(\vec{X_n}) = p_L \cdot \|X_n\|_s + p_D \sum_{i,j} \|T_{ij} - C_{ij}\|^2, \tag{3}$$

where $s \in \{0, 1\}$. Adding the second term to the overall fitness can be done both in integer and in continuous optimization mode. A threshold-value as mentioned above can also be used to let the user control how close a solution must match the original image to be considered as valid.

Thus, an objective quality metric has been defined for determining how good or similar a solution is, and a smaller number encodes a better solution. After having defined the fitness function, the general procedure of the algorithm follows the schema of a typical genetic algorithm. In each generation, new solutions are processed and ranked according to their fitness values. The best solutions are kept by the principle of elitism, while the others are modified in order to gain better solutions from generation to generation. The algorithm would have to find the settings which result in the smallest number representing the quality of a solution. Mathematically speaking, the whole problem can be seen as an optimization problem, where a global minimum has to be found among all possible solutions:

$$\min(f(\vec{X_n})), \quad \forall \vec{X_n} \in \mathbb{R}^n. \tag{4}$$

4 LuxRender

LuxRender is a physically based, state-of-the-art open-source software renderer based on Pharr's and Humphrey's physically based renderer, PBRT [10], which was developed for educational and academic use. *LuxRender* is a stand-alone renderer and not a modeling software. Thus the creation of scenes and models has to be done in other software, and then they have to be exported for rendering.

In 2007, the creators adapted the original source code to make it suitable for artistic use [7]. *LuxRender* implements different state-of-the-art rendering algorithms and provides features such as different material types for objects, post-processing effects, HDR rendering, film response among many others. The implementation presented in this paper works on level of the "Film" stage depicted in figure 1 and makes use of a feature called *Light Groups*. When modeling a scene with several lights, those lights can be associated with a light group. An arbitrary

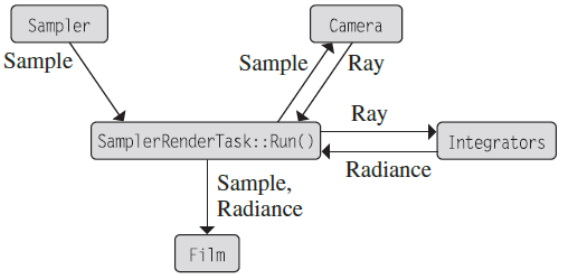


Figure 1: *Basic architecture of PBRT. The Sampler provides the SamplerRendererTask with random samples for BRDF sampling. With that sample, the camera then constructs a ray towards the image plane for the next pixel position and passes it to the Integrator. The integrator calculates the radiance carried along that ray. The collected radiance then gets saved on the film.*
[10]

number of lights - also only a single one - can belong to such a light group. During the rendering process, the light contribution of each group is saved in a separate buffer. Every group also has a intensity and a color temperature, which can be controlled by two parameters. This enables to change the initial light settings in a scene while the rendering is still in process or after it is finished. The user can modify those parameters in the GUI.

5 Results

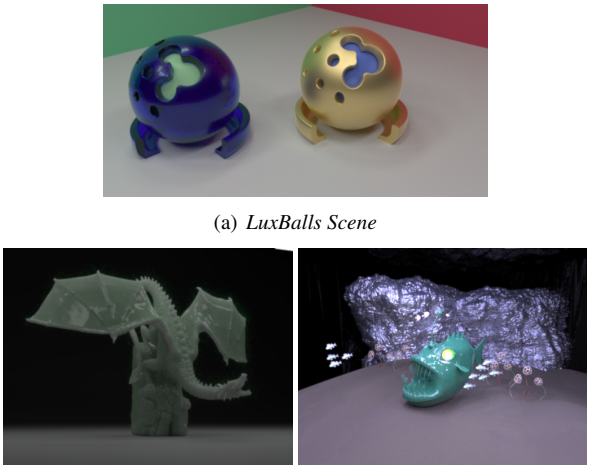
The following section consists of two parts, the first show our results of the empirical test study to verify the assumption that rendering with less light sources increases overall rendering speed. The second part presents the results achieved when using the light source cleaner (further referred to as LSC) on three specially designed test scenes. The scenes were all rendered on a computer with an Intel Core i7-2600K CPU @ 3.40GHz (8 (logical) CPUs), 16 GB DDR3 RAM, and an NVIDIA GeForce GTX 560 Ti with 1 GB of memory.

(a) *Cherry scene* (b) *Watch scene*

Figure 2: *Two of the scenes used for comparison of render times depending on amounts of lights used. The results are presented in Table 1.*

5.1 Test scenes with many/less lights

To verify the assumption that render time can be saved when using less lights in a scene, test renderings of five scenes of varying complexity were done. The test scenes where designed to make use of up to 47 light sources. Each scene was rendered twice with different light source setup: once only with a set of light sources that have a contribution to the lighting of the scene, and once with 17 additional light sources that have almost no contribution to the scene. With the exception of the School Corridor scene, which was rendered for half an hour due to its complexity, the test scenes were rendered for 10 minutes each. A ground truth image of each scene was also rendered for one hour.
The resulting images were then compared to the ground truth using the root mean square error metric (Table 1). The scenes where less lights are used always have a smaller difference to the ground truth image than the ones with many lights. When rendering the test scenes longer, the difference between using less or many lights gets clearly visible for the human observer, without using a comparison software.



(a) *LuxBalls Scene*

(b) *Dragon Scene* (c) *Fish Scene*

Figure 3: *Scenes used for testing the algorithm.*

5.2 Results of the LSC

We have used several scenes to test our method. In every scene, each light used was assigned to a separate light group - otherwise the light sources can not be manipulated individually.

First, the integer optimization mode was tested with the scenes that were already used in the empirical study. In those scenes, there were "fake" light sources which had no contribution to the lighting of the particular scene. The assumption was that the "unnecessary" lights should be easy to determine, so when running for enough generations, the algorithm should be able to detect all of them and turn them off. This worked fine for the tested scenes and

Scene	Balls		Dragon		Corridor		Cherry		Watch	
Lights	30	30+17UL	20	20 + 17UL	10	10 + 17UL	31	31 + 17UL	11	11+17UL
Path tracing	1040.44	1148.54	224.41	250.42	4918.52	5250.92	2012.82	2238.13	1246.18	1489.03
Bidirectional	1218.10	1333.16	257.46	275.04	2957.23	3827.03	2567.93	3115.10	1220.64	1887.13
Metropolis	1209.58	1347.72	283.97	304.10	3103.88	4082.38	2541.94	3031.82	1408.88	2342.89

Table 1: *RMSE of the test scenes compared to ground truth image. Each scene was rendered two times, once with a certain amount of light sources and a second time with 17 additional "unnecessary lights" ("UL"), which had no visible contribution to the scene. Rendering time was 10 minutes for each scene except for the Corridor scene, which was rendered for half an hour due to its complexity. The ground truth images were rendered for one hour. The table shows that the scenes with less light sources have a smaller difference to the ground truth image than the scenes with many lights. This verifies the assumption that rendering is more efficient when using less lights in a scene by canceling out lights with almost no contribution.*

should also work for any arbitrary scenes where lights are used which have (almost) no contribution. However, we show that our technique is capable of solving more complex scenarios beyond these trivial cases.

For the testing the continuous optimization mode, three additional scenes were modeled and tested (see Figure 3). Two rather simple cases were constructed to show that the algorithm generally works. The first one consists of two beveled spheres on a pedestal standing in front of a wall. Three area lights - two small lights and one big light - were then arranged in the following way: each one of the small lights is half the size and half the power of the big light, and the lights were positioned so that the two small lights together are covering the big light exactly. Also, the lights are placed exactly at the same height. Figure 4 shows a screenshot of the 3D-view of the scene for better understanding. The assumption was that if the algorithm worked correctly, it should be possible to achieve the same lighting for a scene when turning off the small light sources completely and increasing the intensity of the bigger area lights. This makes a simple test case which translates to a high-dimensional optimization problem, for which we exactly know the analytic solution.
To make the whole scenario more challenging, this basic setup of three lights was copied and pasted into the scene several times, so that 12 of those arrangements (this makes 33 lights in total) are present in the scene.

The second scene features a dragon model illuminated by 50 area lights, positioned pairwise on the same position and height with the same light intensity. So in this scene, it should be possible to turn off at least half of the lights when increasing the intensity of the other half. A 6x3 array of those lights illuminates the dragon from top, and 7 lights from the front. The light setup is also rather simple here, but the amount of lights is already high. Figure 5 depicts a screenshot again for better understanding.

The third scene which shows an angler fish is the most complex setup featuring 100 light sources. The majority are blue area lights, and there are also point lights of no contribution hidden behind the big stone wall. The algorithm should be able to both turn off many of the point lights and reduce the amount of area lights also significantly. Contrary to the previous scenes, the area lights are positioned arbitrarily instead of being arranged in a special way. This was done to simulate a scene of practical interest as it occurs in film production.

The initial images were rendered with bidirectional path tracing, and the algorithm was run on each of them several times for 100, 500 and 1.000 generations with 15 chromosomes each and different weighting parameters. At the beginning, some runs which last only 100 generations were made several times to ensure that the algorithm works "right" and gives similar results on each run. Figures 6, 7 and 8 show the result images on the $l1$-norm of the light vector at several stages of the optimization process.

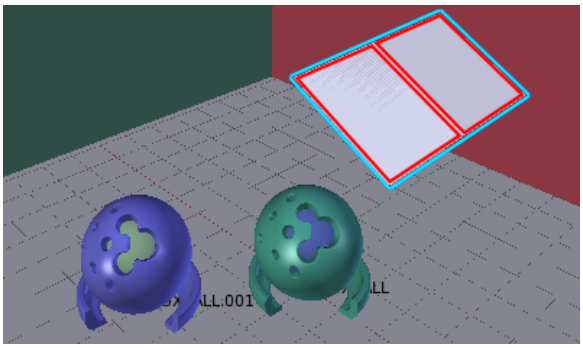


Figure 4: *Example for the light setup in the Luxballs scene. The cyan colored rectangle marks the big area light, while the two red ones mark where the two small area lights are located. The small lights together cover the same area as the big one and are placed exactly at the same height. 12 of those arrangements are present in the scene.*

6 Conclusion and discussion

We presented a light source minimization technique to provide a solution for reducing the overall amount of light sources used in a scene by applying a genetic algorithm to a multivariate optimization problem. A definitive strength is the simplicity of the concept and its general applicability. Although for this paper the implementation was done

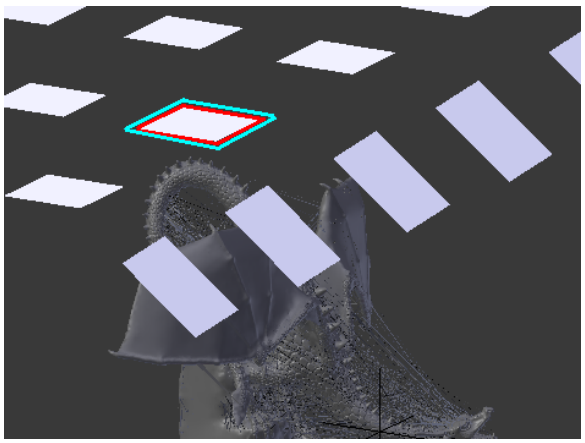


Figure 5: *Example for the light setup in the Dragon scene. The cyan and red rectangles exemplary mark two area lights, which are placed exactly at the same position and have the same intensity.*

in *LuxRender*, this technique can be implemented in any photorealistic rendering engine as long as there is a mechanism that stores the contributions of light sources at different locations. We demonstrated that our technique both passes on scenes with known analytic solutions and also works well on scenes of practical interest. We note that as we are using unbiased and consistent rendering algorithms, it is possible to reduce the number of light sources while the rendering is still in progress.

Another issue is that a global metric is used to measure similarity between two images. As the algorithm basically does a pixel-wise comparison, images with local extrema may impose problems due to the omittance of small local features. Using mean squared error metric instead of the simple difference between the pictures indeed helps, but still there is no means to explicitly consider "important" pixels like highlights or shadows. One possible improvement would be to let the user interactively pre-define which local effects are important and should be kept after optimization.

Our proposed technique is simple to implement, and offers a significant speedup in the execution time of the rendering step in difficult lighting scenarios with a vast amount of light sources.

7 Acknowledgements

We would like to thank Kai Schwebke for providing the LuxTime, Andreas Burmberger for the Cherry Splash, Simon Wendsche for the School Corridor and Peter Sandbacka for the Jade Dragon scene and the LuxRender community for the LuxBall model. The fish scene was created by using various models, we like to thank Paul aka. MajorNightmare for the angler fish, Peter Sandbacka for the sea weed and Chris Monson for the seabed.

References

[1] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[2] Per H. Christensen, David M. Laur, Julian Fong, Wayne L. Wooten, and Dana Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *In Eurographics 2003*, pages 543–552. Blackwell Publishers, 2003.

[3] António Cardoso Costa, António Augusto de Sousa, and Fernando Nunes Ferreira. Lighting design: A goal based approach using optimisation. In *Rendering Techniques*, pages 317–328, 1999.

[4] Christophe Hery and Ryusuke Villemin. Physically based lighting at Pixar, 2013. Accessed: 2013-12-04.

[5] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[6] John K. Kawai, James S. Painter, and Michael F. Cohen. Radioptimization: Goal based rendering. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 147–154, New York, NY, USA, 1993. ACM.

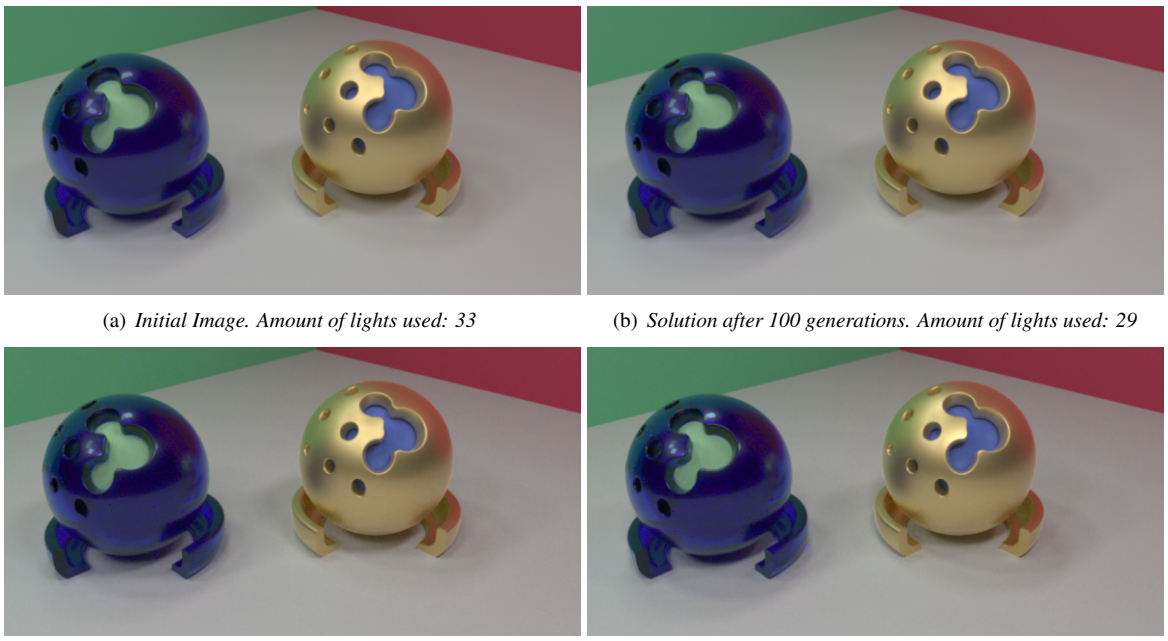
[7] LuxRenderProject. http://www.luxrender.net. Accessed: 2013-12-30.

[8] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 389–400, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[9] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[11] Chris Schoeneman, Julie Dorsey, Brian Smits, James Arvo, and Donald Greenberg. Painting with light. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 143–146, New York, NY, USA, 1993. ACM.



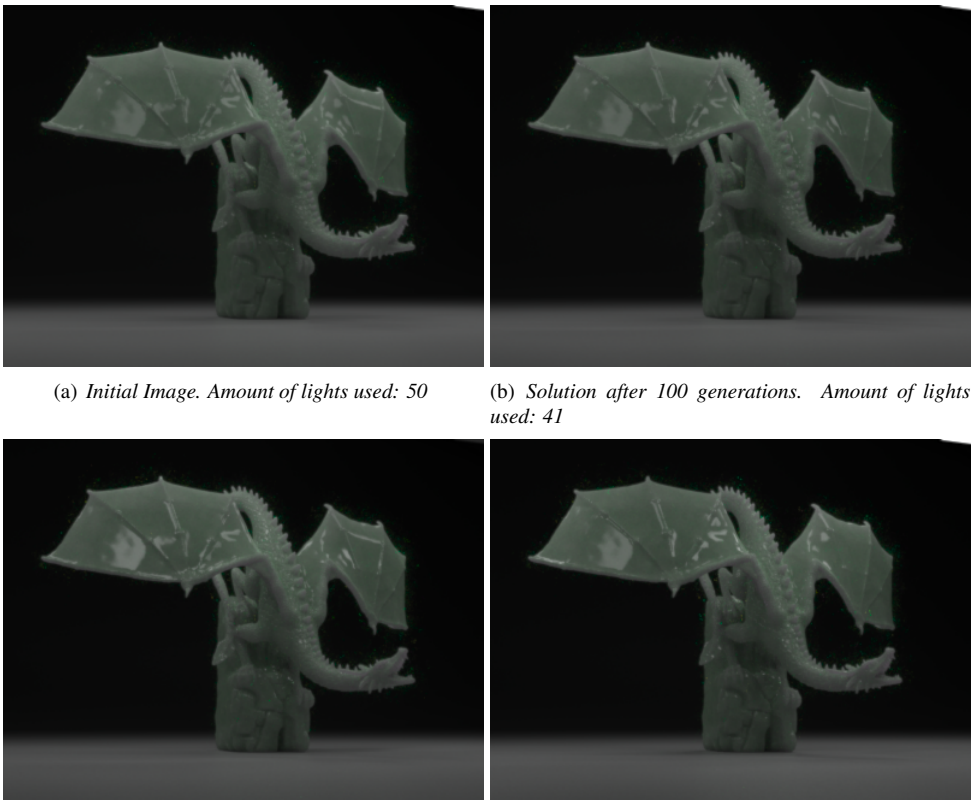
(a) *Initial Image. Amount of lights used: 33*

(b) *Solution after 100 generations. Amount of lights used: 29*

(c) *Solution after 500 generations. Amount of lights used: 11*

(d) *Solution after 1000 generations. Amount of lights used: 12*

Figure 6: *Luxballs Scene. 33 Lights in total.*



(a) *Initial Image. Amount of lights used: 50*

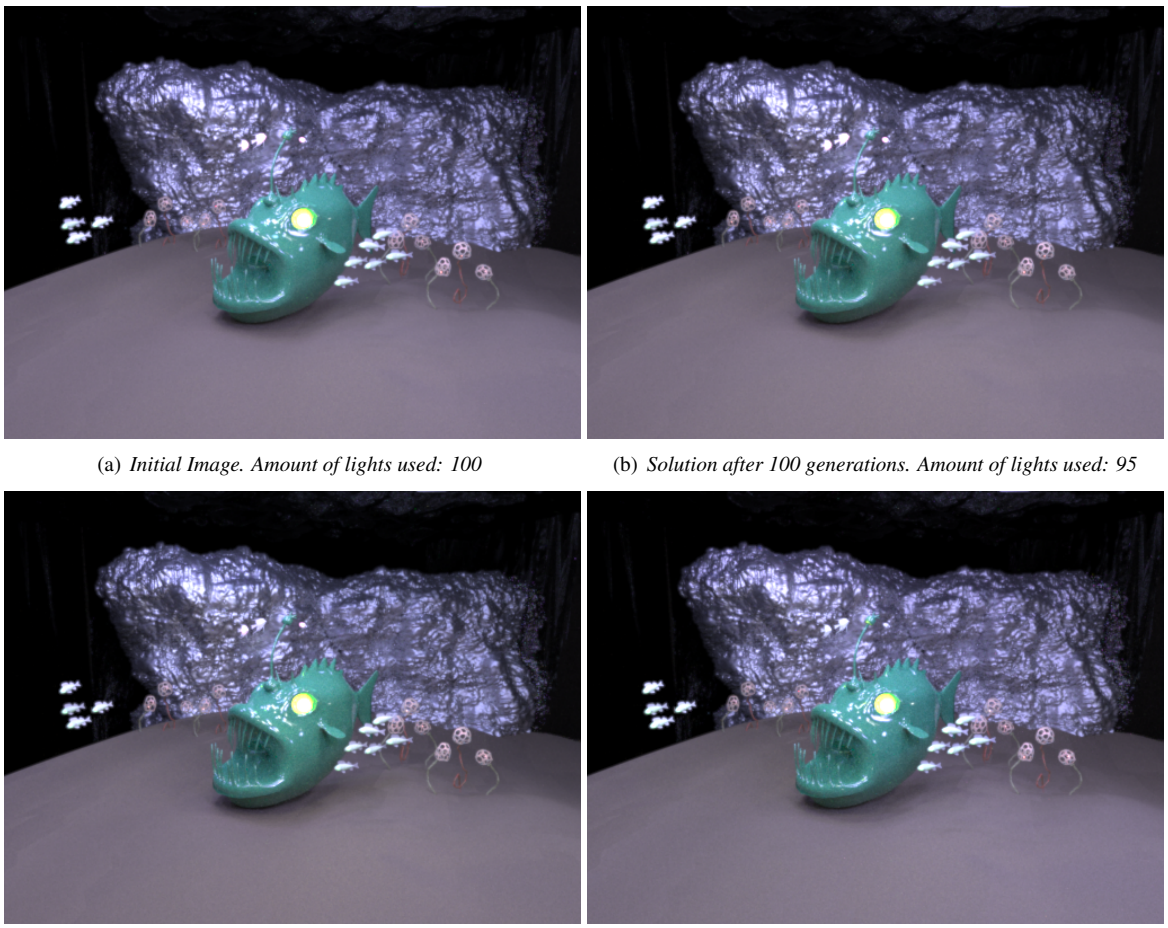
(b) *Solution after 100 generations. Amount of lights used: 41*

(c) *Solution after 500 generations. Amount of lights used:26*

(d) *Solution after 1000 generations. Amount of lights used: 24*

Figure 7: *Dragon Scene. 50 lights in total.*

[12] Karl Sims. Artificial evolution for computer graphics. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, pages 319–328, New York, NY,



(a) *Initial Image. Amount of lights used: 100*

(b) *Solution after 100 generations. Amount of lights used: 95*

(c) *Solution after 500 generations. Amount of lights used: 70*

(d) *Solution after 1000 generations. Amount of lights used: 58*

Figure 8: *Fish scene. 100 lights in total.*

USA, 1991. ACM.

[13] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. Lightcuts: A scalable approach to illumination. *ACM Trans. Graph.*, 24(3):1098–1107, July 2005.

Comparative Evaluation of Photon Mapping Implementations

Tomáš Lysek*
Supervised by: Pavel Zemčík†

Department of Computer Graphics and Multimedia
University of Technology
Brno / Czech Republic

Abstract

The paper focuses on the photon mapping method, which is one of the global illumination methods used in computer graphics. The paper presents a short summary of the photon mapping method and proposes decomposition of photon mapping into a set of simpler algorithms. Each of these algorithms is evaluated in the experimental implementation in order to identify bottleneck(s) in the method. The results of the experimental evaluation are presented in the paper and some suggestions regarding alternative implementation and/or optimization of the computationally expensive parts are presented as well. Finally, the paper presents the results of some of the optimization and draws conclusions.

Keywords: photon mapping, global illumination, kd tree, nearest neighbors

1 Introduction

Photon mapping is one of the advanced rendering methods, which belong to the global illumination methods. Photon mapping is capable of creation of images using advanced photorealistic elements, such as indirect illuminaton and causitcs. Among the global illuminaton tehniques, photon mapping is one of the fastest one which is able to render highly photoroealistic images. Recently, new techniques for acceleration of computer graphics were discovered. For this reason, it is interesing to test these techniques and algorithms with photon mapping.

Photon mapping was first introduced by Henrik Wann Jensen in 1996 [6]. Jensen also wrote a great book [7] about photon mapping where he is comparing photon mapping with other global illumiation methods and presents advanced techniques in photon mapping such as subsurface scattering or rendering participating media. Many extensions to Photon mapping exists, like Progressive Photon mapping [4] and Stochastic Progressive Photon Mapping [3].

Many researchers worked on fast ray-triangle intersection. Good results were achieved by Wald [11] or Shevtstov [10]. The currently best performance has the new method by Havel [5]. In his paper, very good comparison on ray-triangle intersection methods are shown.

Kd-tree was first presented by J. L. Bentley in 1975 [2]. Using KD-tree with triangles is a little more complicated then using it with points. For this reason, complicated heuristic must be performed. Paper by Wald [12] and Havran is dedicated to fast creation of KD-tree on triangles with complicated heuristics. The paper by Zhou [13] shows even more speedup of KD-tree construction using GPGPU.

For fast nearest neighbor searching, it is possible to combine clasical nearest neighbor searching with aproximate searching. The aproximate searching was first presented by Area and Mount [1] in 2000.

In the presented work, the main focus was on the above techniques that were examined, measured on real photon mapping datasets, the best combination was proposed In order to achieve the fastest solution.

2 Photon Mapping

Photon mapping is a two-pass rendering method. It was introduced by Henrik Wann Jensen in 1996 [6]. It is based on aproximation of rendering equation [8] by calculating the incoming radiance of the selected point, where local illumination model is computed, through the nearest photons. In photon mapping, the Photon is a bigger particle than photon known from physics, that, which carries a certain amount of light energy (higher than real photon) but its behavior is similar to photon. Before searching for nearest photon, photon map has to be created for the whole scene. Photon map is a set of distributed photons on the scene which represents illumination of the scene. For this reason, photon mapping is two-pass rendering method. In the first pass, photons are emitted from light sources. These photons are propagated through scene and if the photon hits a diffuse surface, the value of the photon energy is stored into the photon map. Consequently, these photons are recursively being sent to the scene with direction based on surface characteristics [7]. The photons, which get propagated through any transparent objects, are creating caustics on diffuse surfaces and these caustic photons are

*xlysek03@stud.fit.vutbr.cz
†zemcik@fit.vutbr.cz

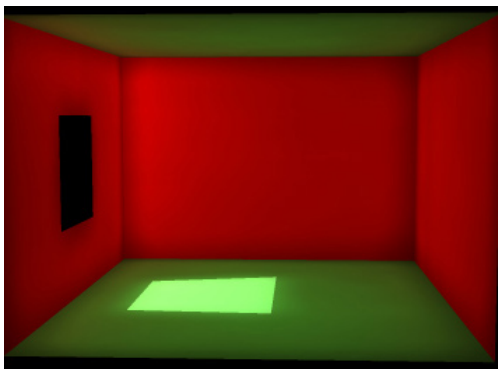


Figure 1: Indirect Illumination

stored in a separate photon map. Caustics are refracted light rays which are concentrated into small areas and they are creating shiny places on diffuse material.

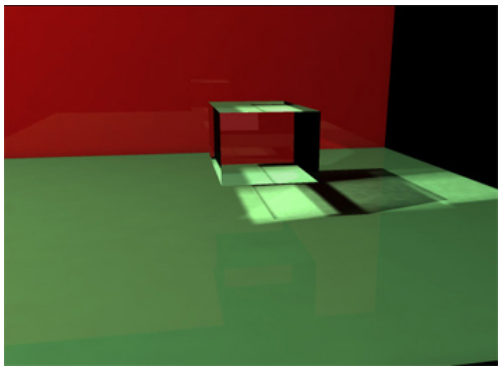


Figure 2: Caustics

Rendering of photon map is possible using several different methods. Probably the most photorealistic results can be achieved by distributed raytracing [7]. Also very good results are achievable by the classic raytracing extended by computing indirect illumination and caustic by nearest photons in photon map [7].

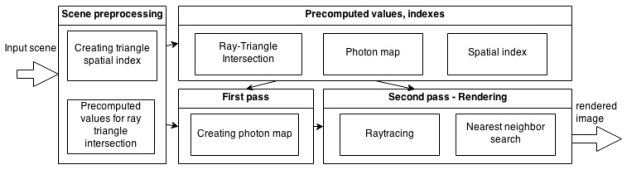


Figure 3: Block diagram

The Photon mapping task is possible to divide into several functional blocks, subtasks, which will be described later and analyzed:

- **Ray-triangle Intersection**
  There are many methods for computing intersection of ray with triangle. In paper Yet Faster Ray-Triangle Intersection [5] is presenting the currently fastest method. Also, during the performed work, it was measured how the another ray-triangle intersection method compares with others. In this comparison, their method has best result. For exploitation of this method, some precalculated values have to be prepared for each triangle.

- **Spatial index (spatial partitioning)**
  Spatial index is data structure which divides space into more smaller subspaces. For each subspaces, all triangles which lie in a subspace are stored in a list connected to the individual subspace. When a ray is being shot through the scene, the ray-triangle tests are performed only on those triangles, which lie in subspaces which intersect with that ray.

  There are many spatial indexes. Performance of each spatial index is highly dependent on scene setup. It is impossible to determine fastest index.

  The most used spatial indexes are octree and KD-tree. Octree is a spatial index, which recursivly divides its space into eight subspaces of the same size. The recursive division is performed to specific level or recursivly dividing is terminated when count of triangles in subspase falls under some specific threshold.

  KD-tree is spatial index which recursively divides space into two subspaces using a dividing plane. This dividing plane is parallel with one of the axes. Several methods exist to determine the dividing plane. The simpler methods include median split on one of the axes (e.g. circularly changed). The more advanced methods use heuristics for determining where the best dividing plane should lie. With these heuristics, is possible to accomplish better results.

  Probably the most used heuritic, used with KD-trees in spatial triangle indexing, is Surface Area Heuristic [13]. This heuristic attempts to maximize the area of subspaces and quantity of triangles in these subspaces.

  Another possible optimization of KD-tree is ropes [9]. Ropes are connections between leafs of the spatial index tree. Using this extension, it is possible to traverse the tree directly through leaf subspaces and avoid slow crawling up and down the tree.

- **Creating photon map**
  In this block, light propagation is simulated from the light sources into the scene and the process results in the photon map. The simulation itself is performed by discrete sampling of light transmission. One sample represents the photon and carries fraction of the light source energy. The light transmission is calculated for example by rejection sampling [7], in which photon is sent from the light source in random direction and then it is propagated through the scene.

  The photons are propagated through the scene similarly to the rays in raytracing. If a photon hits diffuse

surface, energy, direction, and position of the photon are stored. Recursively, another photon from this position is sent further to the scene. The direction of such photon is based on material properties and e.g. if the material is shiny and transparent, two photons are investigated one reflected, second transmitted. For this purpose, an acceleration technique called Russian roulette was created. When the Russian roulette is being used, actions and generation of photon (storage, reflection, refraction) are based on random number with threshold depending on material properties [7].

Photons, which pass through transparent object create caustics. For photorealistic caustic rendering, a large number of photons, which pass through transparent object, is needed. For this purpose, new simulation is started but photons in this simulation go only through transparent objects and save values only to caustic photon map. So, in the end, photon mapping has two photon maps, one for indirect illumination and a second one for caustics.

- **Raytracing**
  For the final rendering of the scene, classical raytracing is used. The values representing indirect illumination and caustics are treated as a local illumination model. These values are obtained by searching N nearest neighbors in photon maps. With increasing N, the quality of the photorealistic results are improved.

- **Finding nearest photons**
  Nearest neighbor search is performed for each computation of local illumination, hence this block is very critical and it has very important role in the rendering performance. The speed of this block is dependent on the size of the photon map and on the number of the neighbors.

  For acceleration of the nearest neighbor search, it is appropriate to use searching index. Many different methods focusing on nearest neighbor search exist. Some of them are available in libraries. One of the interesting ones is ANN Aproximate Nearest Neighbor Library and another one is FLANN Fast Library for Approximate Nearest Neighbors.

  ANN is an older library, this library is used for searching of the KD-trees and BD trees [1]. The FLANN library is used for indexing using the randomized KD-tree. Both libraries are very often used in computer vision for nearest neighbor searching in image features and they are very well optimized for mulidimensional datasets. In our case, analysis in lower dimensionality 3D is needed. Both libraries provide approximate searching search with a small acceptable error is enabled.

3 Experimental Evaluation

The purpose of the experiments is to test selected methods on photon mapping datasets. On these datasets, their performance is evaluated, compared against the other methods, and the method which fit best for photon mapping is then selected.

Experiments which were performed are:

- **Spatial subdivision test** - comparing spatial indexes, octree and KD-tree for acceleration ray-triangle intersection.
- **Creating photon map** compare speed of creating searching indexes on KD-tree and BD-tree in FLANN and ANN libraries
- **Nearest searching - neighbor number** - Comparing nearest neighbor search time with increasing number of neighbors to find.
- **Nearest searching - size of map** - Comparing nearest neighbor searching time with increasing size of photon map.
- **Approximate nearest search** - identification of maximum acceptable error and comparison of the approximate searching times.

All the experiments were performed on laptop with Intel core i7 M620 processor @ 2.67 GHz with 2x 2GB DDR3 RAM 1066Mhz, 7-7-7-20. For compiling, the MSVC 11 (Visual studio 2012) compiler was used. All the measurements were performed on real photon mapping data.

Spatial subdivision test

This test compares speed of spatial indexing methods. In this test, three methods are being compared. First, the naive method with no indexing simply bruteforce method was measured. Then the method is compared to octree and to KD-tree with ropes.

To compare these methods, I created a simple scene with 12140 triangles and performed 100 000 ray-triangle intersection test with this scene. All intersection tests pointing on same spot.

Name	Speed	Precomputing
Naive	47.567s	-
Octree	1.159s	**0.04s**
KD-tree with ropes	**0.453s**	0.16s

Table 1: Comparing spatial indexes

The results of the comparison show that KD-tree with ropes is approximately two times faster than octree. This test also shows that using spatial indexes is indeed efficient and any of the methods outperforms the naive approach.

The KD was 88 times and octree 44 times faster than the naive method.

These test also shows speed of creation of the spatial index. Octree is approximately four times faster than KD-tree. As the spatial index is created only once in this method while the ray-triangle intersections are performed many times - in photon map creating and raytracing. The KD-tree with ropes is generally the best of the tested ones.

Creating photon map

This test compares times of spatial index creation in photon maps depending on the total number of photons in the map. ANN library and FLANN library were used for nearest neighbor searching on multidimensional datasets. From the ANN library, KD-tree index and BD-tree were chosen.

As for the FLANN library, the randomized KD-tree and special single index KD-tree was chosen. Single index KD-tree is optimized for lowerdimensional spaces. This KD-tree is optimized for lower dimensional data and it is called single index.

For this test, I created a simple scene [1] and photon maps with 50k, 100k, 200k, and 500k photons. To generate this amount of photons, I have used photon map block so this test was performed on the real photon mapping data.

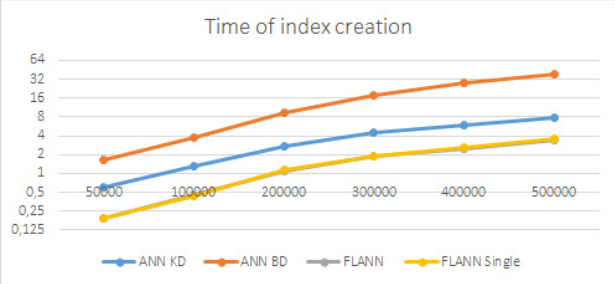


Figure 4: Dependence of map creation time on photon count.

Size of Map	50000	100000	200000	300000	400000	500000
ANN KD	0,591234	1,327776	2,751757	4,408452	6,009744	7,812247
ANN BD	1,660428	3,72888	9,296865	17,34999	27,49657	39,44626
FLANN	**0,187711**	0,458326	**1,066961**	1,878107	**2,476542**	**3,484599**
FLANN Single	0,190411	**0,439525**	1,131965	**1,873107**	2,61855	3,626007

Figure 5: Average times of creating photon map with increasing photons count.

The results show that BD-trees have the worst time of index creation. Both of the FLANN indexes KD-tree and single index have approximately the same time of index creation in fact, KD-tree is little faster than single KD-tree. The ANN KD-tree is approximately five times faster than BD-tree, but two times slower than both of FLANN indexes.

[1] Scene is available at http://lyso.cz/dp/house.zip

Nearest search

This test was performed in order to compare the indexing methods depending on the number of neighbors to search for. The indexing methods for this test were the same as in the previous test - ANN KD-tree + bd tree as well as FLANN randomized KD-tree + single index KD-tree.

The same scene as in the previous tests was used. The photon map with 500 000 photons was created and on this map, the search indices were created. During the testing, the progressively increasing number of nearest neighbor to find were used.

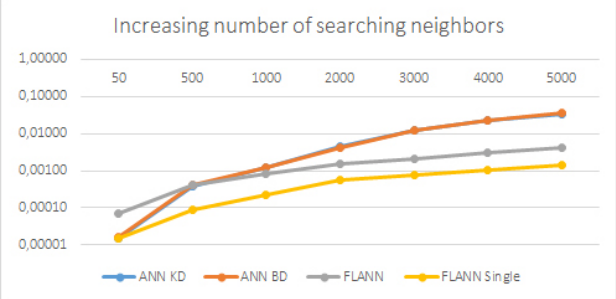


Figure 6: Dependence of searching time on photon count.

N	50	500	1000	2000	3000	4000	5000
ANN KD	0,00001	0,00040	0,00121	0,00452	0,01205	0,02204	0,03394
ANN BD	0,00002	0,00041	0,00121	0,00432	0,01218	0,02281	0,03590
FLANN	0,00007	0,00043	0,00080	0,00151	0,00203	0,00304	0,00410
FLANN Single	0,00001	0,00009	0,00022	0,00054	0,00078	0,00107	0,00141

Figure 7: Average time for searching one photon depending on number of neighbors to search for.

The results show that both ANNs indices and FLANN single index have the best performance for cases in which little number of photons is required to be searched for. However, with the increasing number of photons to search for, ANN is increasingly worse and FLANN KD-tree becomes better than the other two indices.

This test is similar to the previous one but in this case, the number of neighbors is fixed and the size of the photon map is changing. In this experiment, the number of neighbors was set to 5 000. The size of the photon maps ranges from 50 000 to 500 000.

The result shows that increasing size of photon map does not have too big influence on the achieved speed and that the size of the number of photons to search for has large impact on speed.

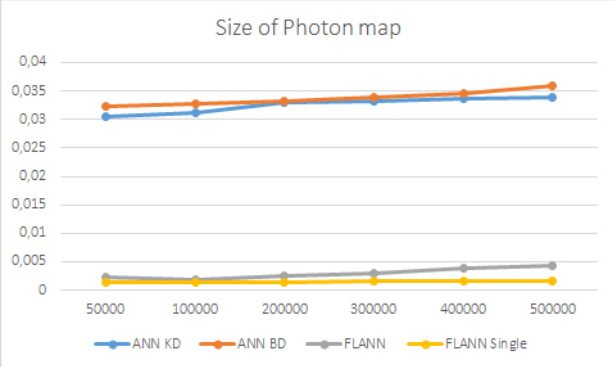


Figure 8: Dependence of searching time on photon map size.

Size of map	50000	100000	200000	300000	400000	500000
ANN KD	0,030481	0,03107	0,032918	0,033227	0,033702	0,03393
ANN BD	0,032248	0,032861	0,033306	0,033909	0,034564	0,035993
FLANN	0,002246	0,001861	0,00254	0,003049	0,003783	0,004382
FLANN Single	0,001426	0,001393	0,001396	0,001489	0,001509	0,001528

Figure 9: Average time for searching one photon depending on number of searching neighbors.

Approximate nearest search

The FLANN and ANN libraries provide also an approximate searching method search in which some error is allowed in the result and which is somewhat faster than the exact case. It should be interesting to find out how much influence the approximate searching has on the quality of rendered images and how much acceleration can be achieved. As the approximate searching leads into worse results in terms of quality, it should be found out how much error is acceptable and then how much it influences the speed.

For this test I created a simple scene, where only the indirect illumination was rendered. This indirect illumination was achieved by search for the nearest photons in the photon map. Photon map size was 500 000 photons and in every calculation of indirect illumination 5 000 photons were used.

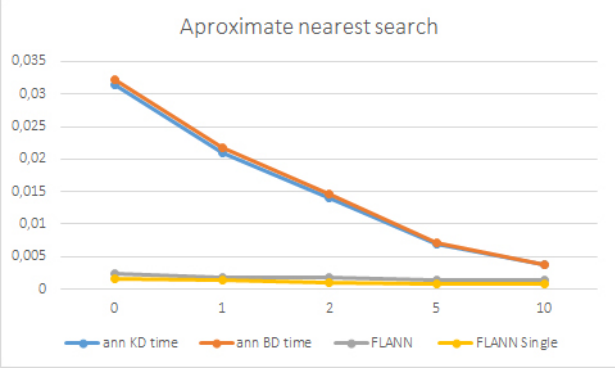


Figure 10: Dependence of searching time on epsilon.

Epsilon	0	1	2	5	10
ann KD time	0,031451	0,020913	0,01414	0,006924	0,003739
ann BD time	0,032258	0,021839	0,014725	0,007142	0,003817
FLANN	0,002356	0,00175	0,001694	0,00145	0,001465
FLANN Single	0,001499	0,001328	0,000981	0,000881	0,000817

Figure 11: Average time for searching one photon depending on epsilon.

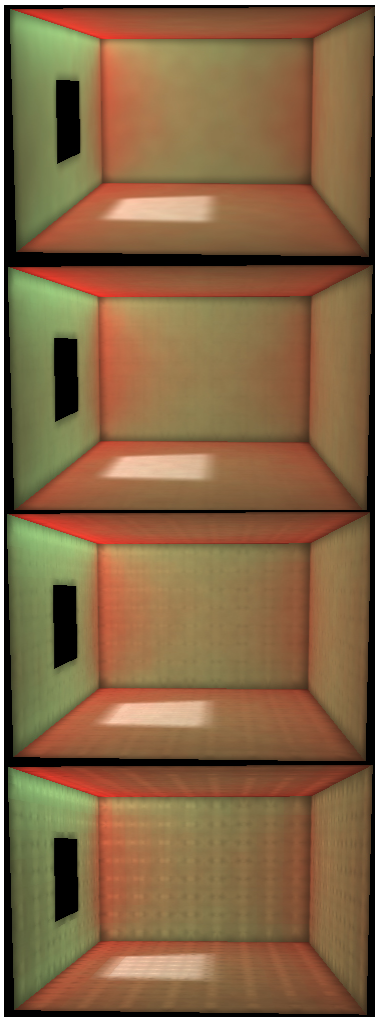


Figure 12: This images shows what influence the allowed error epsilon has on quality of the rendered image. The top images are those with epsilon equal to 0, the second epsilon equal to 1, the third epsilon equal to 2 and the fourth epsilon equal to 4

For measurement acceleration of such approximate searching, another test must be created. For this test, the same simple scene as described above was used. The photon map with 500 000 photons was created and 5 000 photos were used for indirect illumination.

The results show that with the increasing error rate epsilon, the time for searching decreases. The times are decreasing faster when ANN library is used with increasing epsilon but not enough to cause the ANN library to out-

perform the FLANN library. Also, usage of epsilon higher than one has side effects in bad quality of rendered images, as it was mentioned and as it was shown above. For this reason, this approach is generally unusable as it does not produce good enough photorealistic images.

In these experiments, testing of several types of acceleration structures was successfully accomplished. From the results of these experiments, it can be seen what are the best fastest in the application acceleration structures: For spatial index on ray-triangle intersection it is KD-tree with ropes and for acceleration on searching in photon map is the single index KD-tree from FLANN library.

4 Conclusion

In this paper, photon mapping was described along with some selected acceleration techniques. The photon mapping method was subdivided into smaller functional blocks and these blocks were analyzed and their acceleration attempted the acceleration was specifically performed on the slowest blocks of the whole computational process. First experiment was comparing the spatial indices for ray-triangle intersection search. From the results of this text, the KD-tree with ropes was selected as the better one compared to the octree. The experiments with photon map were intended to measure time of creation of the index of a photon map. Several indexing methods were tested and the best performance was accomplished by FLANN indices. Another experiment was performed on evaluation of time for searching for photons in a photon map with increasing number of photons to be searched for. From this experiment single index KD-tree from FLANN library was better. Yet another experiment was performed only to demonstrate that the strongest influence on photon mapping comes from the number of the photons to be searched not size of photon map. Also, when the images are rendered using photon maps, the size of photon map should be bigger. The final experiment was performed on the approximate searching that seemed quite promising. However, this test shows that using approximate searching has side effects in worse quality of rendered image. From all of these experiments on photon maps, it is clear that the single index KD-tree from FLANN library is the best and should be chosen for photon mapping applications. Further work includes more acceleration structures exploration more complex scenes as well as attempt to speedup that could be accomplished by using paralelism for example port photon mapping into GPGPU.

References

[1] Sunil Arya and David M. Mount. Approximate range searching. *Comput. Geom. Theory Appl.*, 17(3-4):135–152, December 2000.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[3] Toshiya Hachisuka and Henrik Wann Jensen. Stochastic progressive photon mapping. *ACM Trans. Graph.*, 28(5):141:1–141:8, December 2009.

[4] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8, December 2008.

[5] Jiri Havel and Adam Herout. Yet faster ray-triangle intersection (using sse4). *IEEE Transactions on Visualization and Computer Graphics*, 16(3):434–438, 2010.

[6] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.

[7] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[8] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[9] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

[10] Maxim Shevtsov, Alexei Soupikov, and Er Kapustin. Ray-triangle intersection algorithm for modern cpu architectures. In *in Proceedings of GraphiCon 2007*, pages 33–39.

[11] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics roup, Saarland University, 2004.

[12] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, pages 61–70, 2006.

[13] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):126:1–126:11, December 2008.

Geometry Processing

Refining Procedures on Mesh via Algebraic Fitting

Tibor Stanko*
Supervised by: doc. RNDr. Pavel Chalmoviansk´y, PhD.†

Faculty of Mathematics, Physics and Informatics
Comenius University
Bratislava / Slovakia

Abstract

A new nonlinear refinement algorithm for surfaces is presented in this work. Our scheme operates on triangular meshes and interpolates input data. Each triangle is associated with a small set of neighbouring points and normals. A low degree algebraic surface (quadric) is fitted to this set with respect to the chosen objective function. The new vertex is taken from the computed quadric. Such a setup overcomes the limitations of the linear schemes. Our experiments show the scheme might be capable of reconstructing quadratic surfaces from a coarse approximating mesh. A comparison of the proposed method with the linear schemes is shown, as well as an application to the compression of a large-scale mesh.

Keywords: mesh refinement, subdivision surface, nonlinear scheme, quadric, mesh compression

1 Introduction

The problem of efficient and accurate geometry modelling of solids has been present in computer graphics from the very beginning. A new approach for boundary representation of three-dimensional objects has emerged in the late 1970s. What became known as *subdivision surfaces* is now widely used in domains such as CAGD, geometry modelling for animation, level-of-detail modelling, multiresolution analysis. For more details on subdivision surfaces and related work, see sections 2 and 3.

A novel nonlinear scheme is proposed in our paper. Even though the scheme is nonlinear, it only requires solving a well-formed system of linear equations for each triangle of the subdivided mesh. For details on the method and the implementation, see sections 4 and 5.

In section 6, we experiment with various sets of weights and analyse the influence of the normal vectors on the limit surface generated by our method. We also show how the resulting method can be used to compress triangular mesh obtained from laser scanning. Such a mesh usually consists of large datasets, typically $\sim 10^5 - 10^6$ vertices. Applying our scheme on properly chosen decimation of input mesh, we are able to reconstruct scanned data very accurately.

The proposed scheme can also be used for the reconstruction of quadratic surfaces from a coarse approximating mesh. We provide a demonstration by reconstructing the sphere from the cube.

2 Subdivision Surfaces

Subdivision is a way of representing smooth shapes in computer [1]. The basic idea of subdivision is to define surface $\mathcal{S}$ as a limit of iterative refinement of mesh

$$\mathcal{S} = \lim_{k\to\infty} \mathcal{M}^k, \tag{1}$$

where the mesh $\mathcal{M}^{k+1}$ is obtained by applying set of refinement rules on the mesh $\mathcal{M}^k$, the mesh $\mathcal{M}^0$ is initial.

A subdivision scheme is *interpolating* if the limit surface interpolates the vertices of the initial mesh. Otherwise, the scheme is *approximating*.

Each mesh $\mathcal{M}$ consists of the topological component (vertices, edges, faces) and the geometric component (vertex positions in $\mathbb{R}^3$). Likewise, every subdivision scheme has *topological* step and *geometric* step. In the topological step, the topology of the mesh in the next iteration is determined. New vertices, edges and faces are inserted, and some of the old ones are removed. Typical operations in this step include inserting new vertices and leaving out some of the old, introducing new edges and faces, flipping an edge. In the geometric step, the new positions of the vertices are computed.

Linear schemes use linear combinations of the vertices from the previous iteration to compute the new positions. Consequently, vertices in the arbitrary iteration $\mathcal{M}^k$ (particularly the limit surface $\mathcal{S} = \mathcal{M}^\infty$) can be expressed as linear combinations of initial mesh $\mathcal{M}^0$ in a natural way. For *nonlinear schemes*, this condition does not hold true.

3 Related Work

Early work on the linear refinement of triangular meshes has been done by Loop [2], who designed an approximating scheme. An interpolating scheme was proposed by

*ts@tiborstanko.sk
†pavel.chalmoviansky@fmph.uniba.sk

Dyn et al. [3] and later modified by Zorin et al. [4]. Another approximating scheme was proposed by Kobbelt [5]. For an overview of existing linear schemes, see [6] or [7]. Extensive bibliography on the topic can be found in the latter.

While linear methods for surface refinement have been closely studied in the past decades, nonlinear methods have received little attention. Linear schemes work well in cases when only the positions of vertices are known. The difficulties arise when we try to make use of data coming from derivatives, such as tangent or curvature. Nonlinear schemes seem to be the right mechanism to bridge this gap.

Only a few nonlinear schemes for surface refinement have been introduced so far. Interpolating triangular algorithms were proposed in [8], [9], [10]. The scheme presented in this paper was inspired by the work of Chalmoviansk\'y and Jüttler [11], who introduced a nonlinear circle-preserving algorithm for *curve refinement*.

4 Refinement via Algebraic Fitting

In this paper, we introduce a different approach to nonlinear subdivision of triangular meshes. The basic idea of the proposed method is to look for the new vertices on the quadric surface, which is the best local approximation of the mesh with respect to the chosen objective function. In the text, we talk about *quadric fitting refinement* or simply QFR when referencing our method.

4.1 Topological step

The quadric fitting refinement uses the topological step introduced by Kobbelt [5]. A new vertex is introduced per triangle face and connected to all vertices of the triangle. Old edges are flipped. Figure 1 shows the topological step on the regular grid for better illustration.

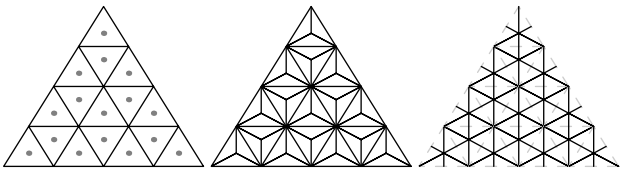


Figure 1: The topological step of Kobbelt's $\sqrt{3}$-subdivision used in our scheme.

4.2 Computing position of the new vertex

Since the proposed scheme is interpolating, the positions of the old vertices remain unchanged. Therefore, the focus of the scheme lies in the computation of the position of the new vertex.

Suppose we want to subdivide the mesh $\mathcal{M}$. Let $\mathcal{V}(\mathcal{M})$ be the set of all vertices of $\mathcal{M}$. We are looking for the position of the new vertex $\mathbf{v}$ introduced in the triangle $T = \mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$. The set $\mathcal{N}_T$ of vertices is called the $m-$neighbourhood of T for some $m \in \mathbb{N}$ if

$$\mathcal{N}_T := \{\mathbf{p} \in \mathcal{V}(\mathcal{M}) \,:\, \mathcal{D}_T(\mathbf{p}) \leq m\} \tag{2}$$

for some $m \in \mathbb{N}$, where

$$\mathcal{D}_T(\mathbf{p}) := \min_{\tilde{\mathbf{v}} \in \mathcal{V}(T)} (\mathcal{D}(\mathbf{p}, \tilde{\mathbf{v}})) \tag{3}$$

is a relative distance of vertex $\mathbf{p}$ from the triangle T, $\mathcal{D}(\mathbf{x},\mathbf{y})$ is a graph distance on $\mathcal{M}$ (number of edges in the shortest path connecting $\mathbf{x}$ and $\mathbf{y}$). We choose m to be the smallest natural number, for which $|\mathcal{N}_T| \geq 9$. Typically, this yields $m = 1$ or $m = 2$. The choice of 9 as the minimal cardinality is justified later in the text (in section 4.3). An example of 1-neighbourhood is shown in figure 2.

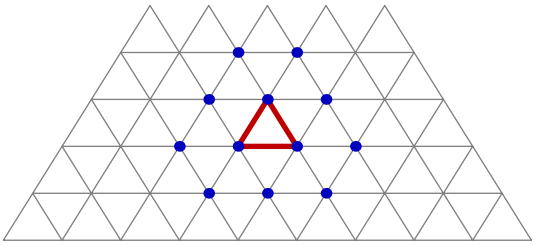


Figure 2: Visualisation of the set $\mathcal{N}_T$ (1-*neighbourhood*) for the triangle T on the regular grid.

The vertex $\mathbf{v}$ is picked out of the quadric surface

$$\mathcal{Q} := \left\{(x,y,z) \in \mathbb{E}^3 \,:\, f(x,y,z) = 0\right\}, \tag{4}$$

where

$$\begin{aligned} f(x,y,z) = a_{11}x^2 + a_{22}y^2 + a_{33}z^2 + 2a_{12}xy + 2a_{13}xz + \\ + 2a_{23}yz + 2a_{14}x + 2a_{24}y + 2a_{34}z + a_{44} \end{aligned} \tag{5}$$

is an unknown trivariate polynomial with real coefficients. Using matrix notation, the equation (5) is written down to

$$f(\mathbf{x}) = \tilde{\mathbf{x}}^\top \mathbf{A}\, \tilde{\mathbf{x}}, \tag{6}$$

where

$$\tilde{\mathbf{x}} := \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, \; \mathbf{A} := \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{12} & a_{22} & a_{23} & a_{24} \\ a_{13} & a_{23} & a_{33} & a_{34} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{pmatrix},$$

$\tilde{\mathbf{x}}$ stands for the homogenous coordinates of $\mathbf{x} \in \mathbb{E}^3$, $\mathbf{A}$ denotes the symmetric matrix of the coefficients a_{ij} from the equation (5).

4.3 Fitting quadric to vertices

We look for such a quadric $\mathcal{Q}$ that is the best approximation of the mesh $\mathcal{M}$ in a close neighbourhood of the

triangle T. For this purpose, we use the set $\mathcal{N}_\mathrm{T}$ defined in (2). In order to specify $\mathcal{Q}$, exactly 10 unknown coefficients $\{a_{ij},\, 1 \le i \le j \le 4\}$ need to be computed up to a non-zero multiple. This clarifies the required condition on the cardinality of $\mathcal{N}_\mathrm{T}$.

To improve the reading comprehension of this section, we use the following notation for gradient operators:

$$\nabla^\mathrm{a} f = \begin{pmatrix} \frac{\partial f}{\partial a_{11}} & \frac{\partial f}{\partial a_{22}} & \frac{\partial f}{\partial a_{33}} & \frac{\partial f}{\partial a_{12}} & \cdots & \frac{\partial f}{\partial a_{44}} \end{pmatrix} \tag{7}$$

denotes gradient with respect to variables $a_{11},\ldots,a_{44}$, while

$$\nabla^\mathrm{x} f = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \end{pmatrix} \tag{8}$$

is the gradient with respect to x,y,z.

Now, suppose $\mathcal{N}_\mathrm{T} = \{\mathbf{p}_i,\, i=1,\ldots,n\}$, where $\mathbf{p}_i = (x_i,y_i,z_i)$. Ideally, $\mathcal{Q}$ interpolates $\mathcal{N}_\mathrm{T}$, meaning f vanishes in every point from $\mathcal{N}_\mathrm{T}$. In general case though, such an interpolation cannot be guaranteed. Instead, we compute the vector

$$\vec{a} \coloneqq \begin{pmatrix} a_{11} & \cdots & a_{44} \end{pmatrix}^\top \tag{9}$$

of the unknown parameters such that the *objective function*

$$\mathcal{F}(a_{11},\ldots,a_{44}) = \sum_{i=1}^{n} \tilde{\mathrm{w}}_\mathrm{i}\, f^2(\mathbf{p}_i) + \hat{\mathrm{w}}_\mathrm{i}\, \|\nabla^\mathrm{x} f(\mathbf{p}_i) - \vec{\mathbf{n}}_i\|^2 \tag{10}$$

is minimized and

$$\vec{a}_\mathrm{min} = \operatorname*{arg\,min}_{a_{11},\ldots,a_{44}} \mathcal{F}(a_{11},\ldots,a_{44}). \tag{11}$$

In (10), the vector $\vec{\mathbf{n}}_i = (\hat{x}_i,\hat{y}_i,\hat{z}_i)^\top$ denotes the normal vector of $\mathcal{M}$ at the vertex $\mathbf{p}_i$. The scalars $\tilde{\mathrm{w}}_\mathrm{i}, \hat{\mathrm{w}}_\mathrm{i} > 0$ are the associated real weights, which are specified later in the text (in section 6.1).

The necessary conditions for minima of the function $\mathcal{F}$ give

$$\nabla^\mathrm{a} \mathcal{F}(\vec{a}) = \vec{0}, \tag{12}$$

a system of linear equations

$$\frac{\partial \mathcal{F}}{\partial a_{ij}}(a_{11},\ldots,a_{44}) = 0, \quad 1 \le i \le j \le 4. \tag{13}$$

If we denote

$$\tilde{\mathcal{F}}(\vec{a}) = \sum_{i=1}^{n} \tilde{\mathrm{w}}_\mathrm{i}\, f^2(\mathbf{p}_i), \tag{14}$$

$$\hat{\mathcal{F}}(\vec{a}) = \sum_{i=1}^{n} \hat{\mathrm{w}}_\mathrm{i}\, \|\nabla^\mathrm{x} f(\mathbf{p}_i) - \vec{\mathbf{n}}_i\|^2, \tag{15}$$

then $\mathcal{F} = \tilde{\mathcal{F}} + \hat{\mathcal{F}}$. Therefore,

$$\nabla^\mathrm{a} \mathcal{F} = \nabla^\mathrm{a} \tilde{\mathcal{F}} + \nabla^\mathrm{a} \hat{\mathcal{F}}. \tag{16}$$

Every vertex $\mathbf{p}_i$ contributes to the objective function in two parts. The function $\tilde{\mathcal{F}}$ measures the distance of $\mathbf{p}_i$ to the quadric $\mathcal{Q}$, weighted by $\tilde{\mathrm{w}}_\mathrm{i}$. The function $\hat{\mathcal{F}}$ measures the deviation of the computed normal from the prescribed normal (at $\mathbf{p}_i$), weighted by $\hat{\mathrm{w}}_\mathrm{i}$.

Let us have a look at the first term on the right side of (16). Denote $\phi_i \coloneqq \phi(\mathbf{p}_i)$, where

$$\phi(\mathbf{p}_i) \coloneqq (\nabla^\mathrm{a} f)(\mathbf{p}_i) = \begin{pmatrix} {x_i}^2 & {y_i}^2 & \cdots & 2z_i & 1 \end{pmatrix}^\top. \tag{17}$$

Using (17) and the fact that $f(\mathbf{p}_i) = \phi_i^\top \vec{a}$, we get

$$\begin{aligned} \nabla^\mathrm{a} \tilde{\mathcal{F}} &= \sum_{i=1}^{n} \tilde{\mathrm{w}}_\mathrm{i}\, 2\,(\nabla^\mathrm{a} f)(\mathbf{p}_i)\, f(\mathbf{p}_i) = \\ &= 2\sum_{i=1}^{n} \tilde{\mathrm{w}}_\mathrm{i}\, \phi_i\, \phi_i^\top \vec{a} = 2\sum_{i=1}^{n} \tilde{\mathrm{w}}_\mathrm{i}\, \Phi_i\, \vec{a} = 2\,\Phi\,\vec{a}. \end{aligned} \tag{18}$$

Here, we have used the notation $\Phi_i \coloneqq \phi_i \phi_i^\top$ and $\Phi \coloneqq \sum_{i=1}^{n} \tilde{\mathrm{w}}_\mathrm{i}\, \Phi_i$ for the corresponding matrices.

Now, we analyse the second term on the right side of (16). First, the gradient of f with respect to x,y,z is computed

$$\nabla^\mathrm{x} f(\mathbf{p}_i) = 2 \begin{pmatrix} a_{11}x_i + a_{12}y_i + a_{13}z_i + a_{14} \\ a_{12}x_i + a_{22}y_i + a_{23}z_i + a_{24} \\ a_{13}x_i + a_{23}y_i + a_{33}z_i + a_{34} \end{pmatrix} =: \begin{pmatrix} \alpha_i \\ \beta_i \\ \gamma_i \end{pmatrix}, \tag{19}$$

where $\alpha_i, \beta_i, \gamma_i$ are dependent on $\vec{a}$. Recall the coordinates of normal $\vec{\mathbf{n}}_i$ at the vertex $\mathbf{p}_i$ are $(\hat{x}_i,\hat{y}_i,\hat{z}_i)$. Consequently,

$$\|\nabla^\mathrm{x} f(\mathbf{p}_i) - \vec{\mathbf{n}}_i\|^2 = (\alpha_i - \hat{x}_i)^2 + (\beta_i - \hat{y}_i)^2 + (\gamma_i - \hat{z}_i)^2. \tag{20}$$

Applying the gradient operator ∇^a on (20), we have

$$\begin{aligned} &\nabla^\mathrm{a} \left(\|\nabla^\mathrm{x} f(\mathbf{p}_i) - \vec{\mathbf{n}}_i\|^2 \right) = \\ &= \nabla^\mathrm{a} \left((\alpha_i - \hat{x}_i)^2 + (\beta_i - \hat{y}_i)^2 + (\gamma_i - \hat{z}_i)^2 \right) = \\ &= 2\,(\alpha_i - \hat{x}_i)\,\nabla^\mathrm{a}\alpha_i + 2\,(\beta_i - \hat{y}_i)\,\nabla^\mathrm{a}\beta_i + 2\,(\gamma_i - \hat{z}_i)\,\nabla^\mathrm{a}\gamma_i. \end{aligned} \tag{21}$$

Note that applying ∇^a on α_i, β_i and γ_i, we get the vectors

$$\begin{aligned} \nabla^\mathrm{a}\alpha_i &= 2\begin{pmatrix} x_i & 0 & 0 & y_i & z_i & 0 & 1 & 0 & 0 & 0 \end{pmatrix}^\top, \\ \nabla^\mathrm{a}\beta_i &= 2\begin{pmatrix} 0 & y_i & 0 & x_i & 0 & z_i & 0 & 1 & 0 & 0 \end{pmatrix}^\top, \\ \nabla^\mathrm{a}\gamma_i &= 2\begin{pmatrix} 0 & 0 & z_i & 0 & x_i & y_i & 0 & 0 & 1 & 0 \end{pmatrix}^\top. \end{aligned} \tag{22}$$

Each of these vectors has only four non-zero coordinates. Plugging (22) into (21) and substituting into (15) yields

$$\begin{aligned} \nabla^\mathrm{a} \hat{\mathcal{F}}(\vec{a}) &= \sum_{i=1}^{n} 4\, \hat{\mathrm{w}}_\mathrm{i} \begin{pmatrix} x_i\,(\alpha_i - \hat{x}_i) \\ y_i\,(\beta_i - \hat{y}_i) \\ z_i\,(\gamma_i - \hat{z}_i) \\ (\alpha_i - \hat{x}_i)\,y_i + (\beta_i - \hat{y}_i)\,x_i \\ (\alpha_i - \hat{x}_i)\,z_i + (\gamma_i - \hat{z}_i)\,x_i \\ (\beta_i - \hat{y}_i)\,z_i + (\gamma_i - \hat{z}_i)\,y_i \\ \alpha_i - \hat{x}_i \\ \beta_i - \hat{y}_i \\ \gamma_i - \hat{z}_i \\ 0 \end{pmatrix} \\ &= 4\sum_{i=1}^{n} \hat{\mathrm{w}}_\mathrm{i}\,(2\,\Psi_i\,\vec{a} - \Omega_i), \end{aligned} \tag{23}$$

where

$$\Psi_i := \begin{pmatrix} x_i^2 & 0 & 0 & x_i y_i & x_i z_i & 0 & x_i & 0 & 0 & 0 \\ 0 & y_i^2 & 0 & x_i y_i & 0 & y_i z_i & 0 & y_i & 0 & 0 \\ 0 & 0 & z_i^2 & 0 & x_i z_i & y_i z_i & 0 & 0 & z_i & 0 \\ x_i y_i & x_i y_i & 0 & x_i^2+y_i^2 & y_i z_i & x_i z_i & y_i & x_i & 0 & 0 \\ x_i z_i & 0 & x_i z_i & y_i z_i & x_i^2+z_i^2 & x_i y_i & z_i & 0 & x_i & 0 \\ 0 & y_i z_i & y_i z_i & x_i z_i & x_i y_i & y_i^2+z_i^2 & 0 & z_i & y_i & 0 \\ x_i & 0 & 0 & y_i & z_i & 0 & 1 & 0 & 0 & 0 \\ 0 & y_i & 0 & x_i & 0 & z_i & 0 & 1 & 0 & 0 \\ 0 & 0 & z_i & 0 & x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

$$\Omega_i := \left(x_i\hat{x}_i,\ y_i\hat{y}_i,\ z_i\hat{z}_i,\ x_i\hat{y}_i+y_i\hat{x}_i,\ x_i\hat{z}_i+z_i\hat{x}_i,\ y_i\hat{z}_i+z_i\hat{y}_i,\ \hat{x}_i,\ \hat{y}_i,\ \hat{z}_i,\ 0\right)^\top. \tag{24}$$

Introducing the notation

$$\Psi := \sum_{i=1}^{n} \hat{w}_i\, \Psi_i, \quad \Omega := \sum_{i=1}^{n} \hat{w}_i\, \Omega_i, \tag{25}$$

the equation (23) becomes

$$\nabla^a \hat{\mathcal{F}}(\vec{a}) = 8\Psi\,\vec{a} - 4\,\Omega. \tag{26}$$

Using the equations (12), (18) and (26), we obtain the desired system of linear equations in matrix form

$$\begin{aligned} \nabla^a \mathcal{F} &= 2\,\Phi\,\vec{a} + 8\Psi\,\vec{a} - 4\,\Omega = \\ &= (2\Phi + 8\Psi)\,\vec{a} - 4\,\Omega = \Gamma - 4\Omega = 0, \end{aligned} \tag{27}$$

with its explicit solution

$$\vec{a}_{\min} = 4\,\Gamma^{-1}\,\Omega, \tag{28}$$

provided $\Gamma = 2\Phi + 8\Psi$ is an invertible matrix.

4.4 Picking the new vertex

After the quadric $\mathcal{Q}$ has been found by solving the system (27), we are able to compute the coordinates of the new vertex $\mathbf{v}$. Denote

$$\mathbf{b}_T := \frac{\mathbf{v}_0 + \mathbf{v}_1 + \mathbf{v}_2}{3} \tag{29}$$

to be the barycenter of the triangle T.

4.4.1 Intersection of normal line and quadric

Our first approach is to find the position of $\mathbf{v}$ as the intersection of quadric $\mathcal{Q}$ and the normal line $\bar{n}$ of the triangle T. The line $\bar{n}$ is defined parametrically as

$$\bar{n} \equiv \mathbf{b}_T + t\vec{\mathbf{n}}_T\,,\ t \in \mathbb{R}. \tag{30}$$

The vector $\vec{\mathbf{n}}_T$ is defined as the unit normal of the plane determined by the vertices of T (modulo the vector signum), see fig. 3. Denote the intersection of $\mathcal{Q}$ and $\bar{n}$

$$\mathbf{v}_T = \mathbf{b}_T + t_0\,\vec{\mathbf{n}}_T, \tag{31}$$

or, coordinate-wise,

$$\begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} = \begin{pmatrix} x_b \\ y_b \\ z_b \end{pmatrix} + t_0 \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} \tag{32}$$

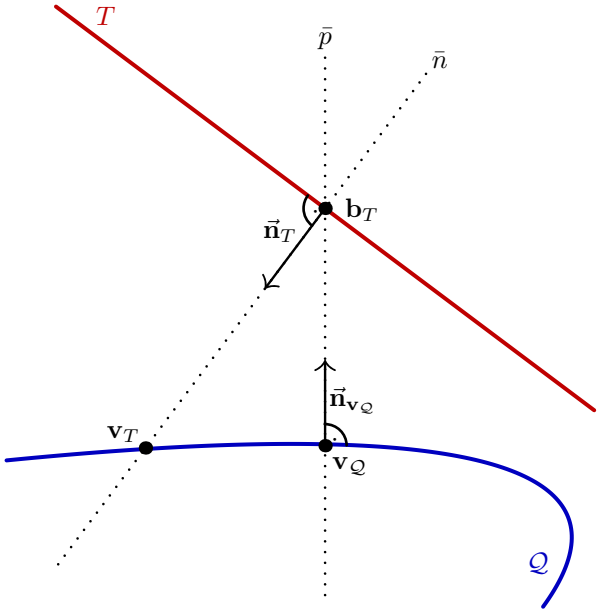


Figure 3: Schematic comparison of the two approaches for picking the new vertex from the quadric $\mathcal{Q}$. The technique described in section 4.4.1 yields $\mathbf{v}_T$, while the technique from 4.4.2 yields $\mathbf{v}_\mathcal{Q}$.

for some t_0. Plugging (31) into (4), (5), we get

$$a_{11}x_v^2 + a_{22}y_v^2 + \cdots + a_{34}z_v + a_{44} = 0. \tag{33}$$

This leads to the quadratic equation in t_0 of the form

$$At_0^2 + 2Bt_0 + C = 0\,, \tag{34}$$

where the coefficients $A, B, C \in \mathbb{R}$ are

$$\begin{aligned} A = {} & x_n\,(a_{11}x_n + a_{12}y_n + a_{13}z_n) + \\ & y_n\,(a_{12}x_n + a_{22}y_n + a_{23}z_n) + \\ & z_n\,(a_{13}x_n + a_{23}y_n + a_{33}z_n)\,, \end{aligned}$$

$$\begin{aligned} B = {} & x_n\,(a_{11}x_b + a_{12}y_b + a_{13}z_b + a_{14}) + \\ & y_n\,(a_{12}x_b + a_{22}y_b + a_{23}z_b + a_{24}) + \\ & z_n\,(a_{13}x_b + a_{23}y_b + a_{33}z_b + a_{34})\,, \end{aligned}$$

$$C = f\,(x_b, y_b, z_b)\,.$$

Denote the roots of (34) as t_1, t_2. The parameter t_0 is chosen as

$$t_0 = \begin{cases} 0, & \text{if } B^2 - 4AC < 0; \\ t_1, & \text{if } |t_1| < |t_2|; \\ t_2, & \text{otherwise.} \end{cases} \tag{35}$$

If the parameters t_1, t_2 are real, they determine two points on $\bar{n}$. We pick the point which is closer to $\mathbf{b}_T$. If t_1, t_2 are complex, the barycenter $\mathbf{b}_T$ is picked as the new point.

4.4.2 Foot point of barycenter

Although the procedure described in section 4.4.1 is easy to implement, it does not generate optimal choice of the

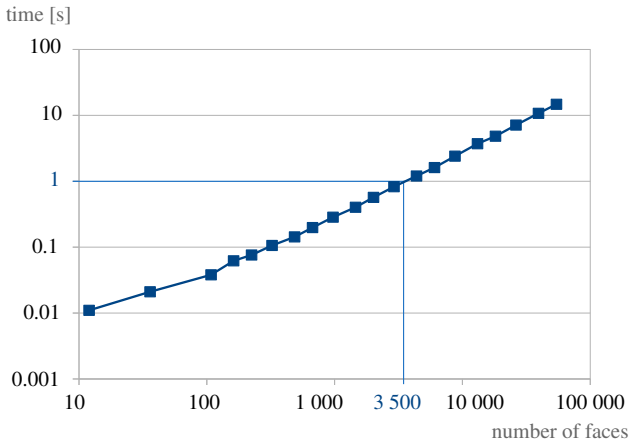


Figure 4: Experimental measurements of the time complexity of our algorithm. In average, we were able to process 3 500 faces per second.

new vertex. In some cases, the intersection of $\mathcal{Q}$ and $\bar{n}$ does not exist or is relatively distant from the mesh. The distant vertices create unwanted local sharpness (spikes). Moreover, if the initial mesh is not closed, the spikes naturally occur around the boundary.

These issues are resolved by picking the new vertex as a foot point of perpendicular line $\bar{p}$ from $\mathbf{b}_T$ onto $\mathcal{Q}$, see figure 3. To find the foot point, we use the algorithm described by Hartmann in [12, section 5.1.2].

5 Implementation

Implementation of our method was done in C++ and compiled under GCC 4.8.1. Both techniques for picking the new vertex from the quadric were implemented. The results in this paper were obtained using the foot point algorithm exclusively.

For mesh manipulation, we decided to use an open-source library *OpenMesh* [13], developed by working group of Leif Kobbelt at RWTH Aachen University. We chose *OpenMesh* for two main reasons:

- ◇ meshes are represented using doubly-connected edge list (DCEL), which allows fast performance of the mesh operations.
- ◇ *OpenMesh* contains application *Subdivider* with built-in framework for subdivision surfaces. *Subdivider* also implements various linear triangular subdivision schemes (Loop, $\sqrt{3}$, Modified Butterfly) overloading abstract base class `SubdividerT`. Simple GUI is provided using Qt and GLUT. User can load and save mesh in popular formats (.obj, .off, .ply) and iteratively apply subdivision operators.

To solve the linear system (27), we used an open-source C++ linear algebra library Armadillo [14].

Computational complexity of the quadric fitting refinement is $\mathcal{O}(F)$, where F is the number of processed faces.

Figure 4 shows the relation between the time needed to perform one iteration of the QFR and the number of triangles in the refined mesh. All the measurements were performed on the PC with Intel Core i7 3517 Ivy Bridge processor running Ubuntu 13.10 Saucy Salamander.

6 Results

6.1 Choosing the weights

Theoretically, any positive real number can be used as a weight $\tilde{w}_i$ of the vertex $\mathbf{p}_i$ or as a weight $\hat{w}_i$ of the normal $\vec{\mathbf{n}}_i$. In practice though, the weights have to be chosen carefully as their choice can influence the result significantly.

The used weights are dependent on the graph distance $\mathcal{D}_T(\mathbf{p}_i) =: \mathcal{D}_T^i$ between the vertex $\mathbf{p}_i \in \mathcal{N}_T$ and the triangle T, see (3). Given the initial values v_i, n_i and factors v_f, n_f, the weights are computed as

$$\tilde{\mathrm{w}}_\mathrm{i} = v_i\, v_f^{\mathcal{D}_T^i}, \quad \hat{\mathrm{w}}_\mathrm{i} = n_i\, n_f^{\mathcal{D}_T^i}. \tag{36}$$

We demonstrate the effect of different sets of weights on the Stanford bunny, see figure 5. The bunny mesh was decimated to 3000 faces and refined using QFR with the initial values and factors

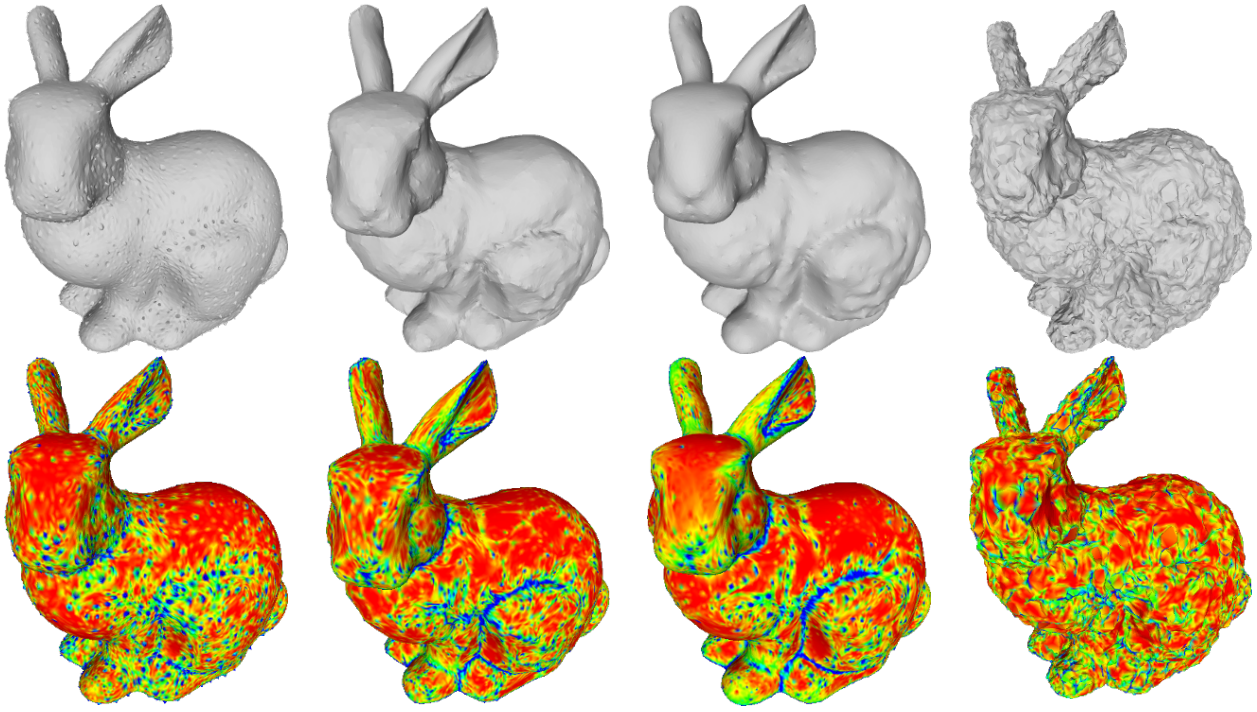
$$v_i = 1, \quad v_f = 1, \quad n_i = 1, \quad n_f = 1; \tag{37a}$$
$$v_i = 1000, v_f = 1, \quad n_i = 0.0001, n_f = 0.0001; \tag{37b}$$
$$v_i = 1000, v_f = 0.0001, n_i = 0.0001, n_f = 0.0001. \tag{37c}$$

It is clear the strategy of taking all data with the same weights as in (37a) does not produce fine results for an irregular mesh such as the Stanford bunny. This is due to the fact that the information carried by normal vectors is very strong and has to be treated gently. Assigning the normals smaller weights as in (37b,c) yields much smoother result. The best results are obtained in (37c), where both $\tilde{\mathrm{w}}_\mathrm{i}, \hat{\mathrm{w}}_\mathrm{i}$ get smaller as the distance from the refined triangle increases.

In our current setup, the weights need to be adjusted case-by-case. One of the possible future improvements of the QFR is to compute the weights algorithmically. The local geometry of the mesh (vertex angles, triangle areas) could be used for this purpose.

6.2 Influence of the normal vectors

The normal vector $\vec{\mathbf{n}}_i$ determines the tangent plane at the vertex $\mathbf{p}_i$. To show how the change in the prescribed normals influences the limit surface, we applied the QFR on the Stanford bunny with the alternative set of vertex normals. This alternative set of normals was generated randomly from a noise function. The results are visualised in figures 5c (original normals) and 5d (random normals). The weights from (37c) were used in both cases. The refined meshes differ dramatically, despite the fact the weights for normals are much smaller comparing to the vertex weights (order of 10^7).



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

Figure 5: (a-c) The Stanford bunny with original normals, refined using the sets of weights from (37a-c). (d) The Stanford bunny with randomly generated normals, refined using the weights from (37c). Bottom part shows the visualisation of the discrete ABS curvature.

6.3 Comparison with linear schemes

To compare the proposed algorithm with the linear schemes, we have used the large-scale mesh of the Venus of Dolní Věstonice. This mesh is a discretized version of the small nude female statuette found in Moravia south of Brno. Dated to 29,000-25,000 BCE, it is considered one of the oldest known pieces of ceramic in the world.

The original Venus mesh (131 114 vertices) was decimated with app. 99% compression rate (1 356 vertices). The decimated mesh was refined four times using the QFR, the $\sqrt{3}$-subdivision, the Modified butterfly and the Loop scheme. For the QFR, we have used the weights $(v_i, v_f) = (1, 0.1), (n_i, n_f) = (0.001, 0.01)$. The one-sided Hausdorff distance was used to measure the error and to compare the refined meshes. For reference, the lengths of the sides of the bounding box of the Venus mesh are $108.4, 31.8$, and 42.8 units.

	2nd iteration			4th iteration		
	max.	**mean**	**RMS**	**max.**	**mean**	**RMS**
QFR	1.945	0.092	0.173	1.945	0.093	0.174
$\sqrt{3}$	1.990	0.167	0.226	2.003	0.174	0.233
MB	1.846	0.083	0.163	1.839	0.084	0.164
Loop	2.001	0.170	0.230	2.003	0.175	0.234

Table 1: Performance of the QFR on the Venus mesh comparing to the linear schemes

The results are visualised in figure 7. The numerical values of maximum, mean and RMS error are summarized in table 1.

Using this setup, we are able to obtain a close approximation of the original mesh. The performance of QFR is comparable to the Modified Butterfly. This is related to the fact that both QFR and Butterfly are interpolating schemes. However, the mesh produced by our method is visually smoother. The meshes generated by the approximating schemes ($\sqrt{3}$-subdivision, Loops) are also smooth, but they lack the details of the mesh produced by the QFR.

6.4 Reconstruction of quadratic surfaces

In the context of our refinement method, quadratic surfaces or *quadrics* are an important tool. Our algorithm can also be used for the reconstruction of quadratic surfaces from a coarse, approximating mesh.

Using the weights $\tilde{w}_i = 1000, \hat{w}_i = 0.0001$, the scheme is capable of reconstructing a close approximation of the sphere from the cube, see figure 6. The initial mesh (cube) is shown after $0, 1, 2, 3$ and 9 iterations, together with the color visualisation of the distance of the densest mesh from the sphere. The red color corresponds to zero distance, blue color corresponds to distance ≥ 0.0025, which is fairly small taking into account the sphere has unit radius. The output is also influenced by the initial triangulation of the cube.

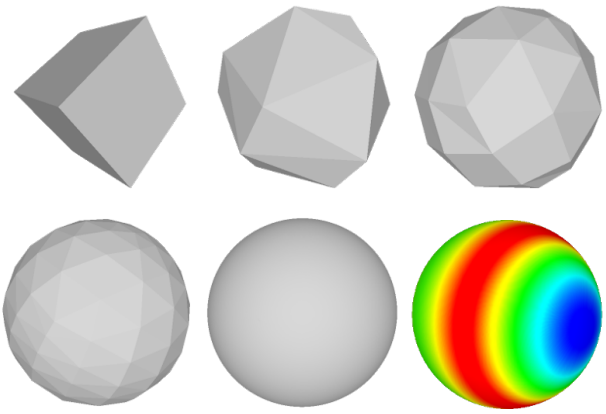


Figure 6: Reconstruction of the sphere $x^2 + y^2 + z^2 = 1$.

In addition to the sphere, the scheme was capable of reconstructing a cylinder, an elliptic paraboloid and a hyperbolic paraboloid. These experimental results allow to make a hypothesis that QFR actually *reproduces* quadratic surfaces. In the future, we want to study this hypothesis from the analytic point of view.

7 Conclusions

We introduce a new approach to nonlinear surface subdivision. While developing the scheme, we encountered problems with spikes, arising in some regions of the mesh and around boundary. These issues are resolved by altering the way the new vertex is picked from the quadric. Although the alternative setup is more complex, it gives more accurate results and is applicable on general input mesh.

In the future, we plan to study the algorithm from the analytical point of view. We want to prove the limit surface is G^1-continuous and confirm the hypothesis about the reproduction of quadratic surfaces. As we have mentioned in section 6.1, we also plan to improve the computation of weights, which should be determined by the local geometry of the mesh.

Even though we assume triangular mesh, the proposed scheme can be extended to quad mesh in a straightforward way. To perform the extension, appropriate topological step has to be chosen.

Acknowledgement

We would like to thank Moravian Museum* and EDICO SK, Inc† for providing the mesh of Venus of Dolní Věstonice we used to test our method. The bunny mesh was kindly provided by the Stanford University Computer Graphics Laboratory‡.

*www.mzm.cz
†www.edico.sk
‡graphics.stanford.edu

References

[1] M. Sabin, *Analysis and Design of Univariate Subdivision Schemes*. Springer Berlin Heidelberg, 2010.

[2] C. Loop, "Smooth subdivision surfaces based on triangles," Master's thesis, University of Utah, August 1987.

[3] N. Dyn, D. Levin, and J. A. Gregory, "A butterfly subdivision scheme for surface interpolation with tension control," *ACM Transactions on Graphics (TOG)*, vol. 9, no. 2, pp. 160–169, 1990.

[4] D. Zorin, P. Schröder, and W. Sweldens, "Interpolating subdivision for meshes with arbitrary topology," in *Proceedings of SIGGRAPH 96, Annual Conference Series*, pp. 189–192, 1996.

[5] L. Kobbelt, "$\sqrt{3}$-subdivision," in *Proceedings of SIGGRAPH 2000, Annual Conference Series*, pp. 103–112, 2000.

[6] T. J. Cashman, "Beyond Catmull–Clark? A survey of advances in subdivision surface methods," *Computer Graphics Forum*, vol. 31, no. 1, pp. 42–61, 2012.

[7] J. Peters and U. Reif, *Subdivision surfaces*. Springer, 2008.

[8] S. Karbacher, S. Seeger, and G. Häusler, "A non-linear subdivision scheme for triangle meshes," in *VMV*, pp. 163–170, 2000.

[9] N. Aspert, T. Ebrahimi, and P. Vandergheynst, "Non-linear subdivision using local spherical coordinates," *Computer Aided Geometric Design*, vol. 20, no. 3, pp. 165–187, 2003.

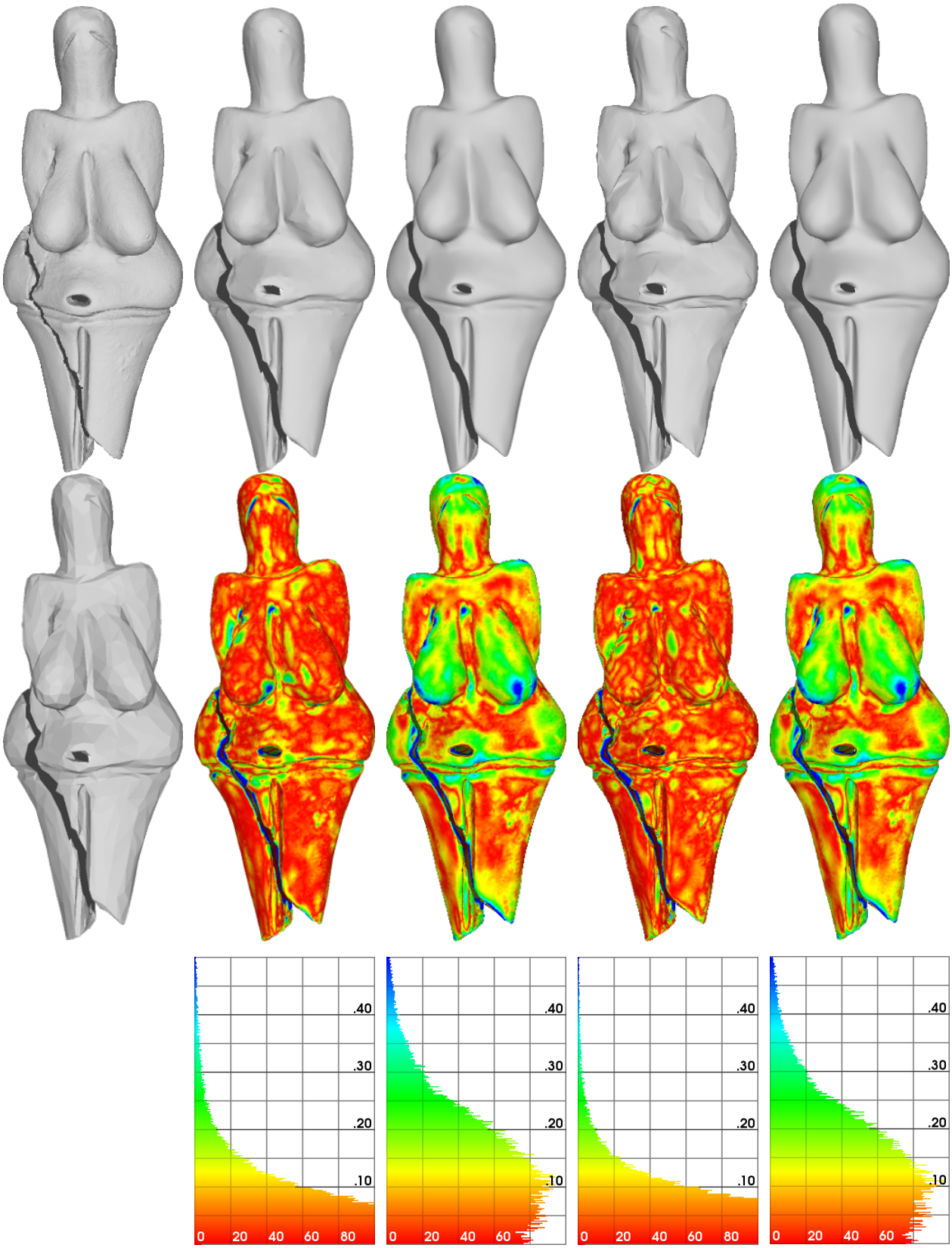
[10] X. Yang, "Surface interpolation of meshes by geometric subdivision," *Computer-Aided Design*, vol. 37, no. 5, pp. 497–508, 2005.

[11] P. Chalmoviansk\'y and B. Jüttler, "A non-linear circle-preserving subdivision scheme," *Advances in Computational Mathematics*, vol. 27, no. 4, pp. 375–400, 2007.

[12] E. Hartmann, "On the curvature of curves and surfaces defined by normalforms," *Computer Aided Geometric Design*, vol. 16, no. 5, pp. 355–376, 1999.

[13] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt, "Openmesh - a generic and efficient polygon mesh data structure," 2002.

[14] C. Sanderson, "Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments," tech. rep., NICTA, 2010.



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

Figure 7: Comparison of our method with the linear triangular schemes. (a) Original and decimated Venus mesh, (b-e) decimated mesh refined with QFR, $\sqrt{3}$, Modified Butterfly and Loop. Top row in (b-e) shows the mesh after four iterations of given scheme, middle row shows the visualisation of the Hausdorff distance of the original mesh from the refined meshes. Bottom row displays the histograms of the Hausdorff distance.

Base Manifold Meshes from Skeletons

Michal Piovarči*
Supervised by: Martin Madaras

Faculty of mathematics physics and informatics
Comenius university
Bratislava / Slovakia

Abstract

We propose an algorithm that generates a base manifold mesh from an input skeleton, based on Skeleton to Quad Dominant Mesh (SQM) algorithm which converts skeletons to meshes composed mainly from quadrilaterals. Each node in skeleton has assigned a sphere with a predefined radius. SQM algorithm first creates branch node polyhedrons for each sphere corresponding to a branch node. These polyhedrons are bridged with quadrilaterals in order to create the final base mesh. We have extended the algorithm to support generation of meshes from cyclic skeletons. We have also generalized skeleton nodes to ellipsoids instead of spheres. Finally, we extended the algorithm to generate meshes from linear skeletons without branching and from skeletons which root node is not a branch node. The generated base mesh is tessellated on GPU for better visual results.

Keywords: skeleton, convert, base mesh, manifold

1 Introduction

Skeletal structures are often used in computer graphics to represent basic topology of a model. This representation allows artists to conveniently animate articulated models, by manipulating key points represented as joints in skeletons. Skeletons corresponding to a model, are often provided by an artist, or extracted directly from the model [1]. Since skeletal structures carry an information about the topology of a model, we could apply a reverse process to skeleton extraction and recover the base mesh represented by a skeleton.

Such base meshes, generated directly from skeletal structures, could be used to ease the modelling of base models of articulated characters. An artist would only design the skeleton of the model and the base mesh would be generated automatically. This technique can also be used to procedurally generate articulated models. A base mesh generated from a supplied skeleton can be augmented with procedurally generated displacement maps in order to generate a complex model.

*michal.piovarci@gmail.com

In Section 2, the state of the art methods used in the area are described. In Section 3, the original SQM algorithm and its drawbacks are discussed. In Section 4, our implementation of base mesh generation is described. In Section 5, our proposed solutions to discussed drawbacks of the original SQM algorithm are presented. Finally, in Section 6 the results of our implementation are presented.

2 Related Work

The most notable algorithms generating base meshes from skeletons are B-mesh [4] by Ji et al. and SQM [2] by J. A. Bærentzen et al. The input for both algorithms is a skeleton with a sphere defined for each node of the skeleton which represents the local geometry of desired output base mesh. Both algorithms present a different way how to approach generation of base meshes from the input skeleton.

The former B-Mesh algorithm firstly generates geometry for paths connecting skeletal branch nodes. These paths are then stitched together at each branch node and the resulting mesh is evolved to better approximate the input skeleton. On the other hand SQM algorithm uses a reverse process. First polyhedrons corresponding to each branch node are generated. The generated polyhedrons are joined together via a tube consisting of quadrilaterals. The resulting mesh is subdivided to increase visual quality.

There are more techniques that generate base meshes but are not limited or used on skeletal structures only. In Solidifying wireframes [7] Srinivasan et al. proposed a method similar to B-Mesh. The proposed method firstly generated mesh corresponding to tubular parts of wireframes. These paths are later joined at branch nodes in a similar manner as in B-Mesh. Although the method is more general than B-Mesh it suffers from the same drawbacks mainly the stitching geometry which produces undesired triangular faces. In a more recent paper Leblanc et al. [6] proposed generating base meshes by iteratively combining blocks into cuboid shapes. Since our algorithm should operate on skeletal structures, by limiting the connectivity of blocks to skeletal structures only, we would lose many of the advantages of the original technique. Base mesh could be also recovered from medial axis transform [8]. However, due to the nature of me-

dial axis transform it is not suitable for editing by artists. Taking in account all the previous drawbacks and that "B-Mesh produces three to four times more irregular vertices than SQM" [2], we have decided to base our algorithm on SQM.

3 Original SQM Algorithm

The algorithm consist of four steps and one preprocessing step:

Preprocessing: Skeleton straightening - serves to simplify step number 3 of the algorithm.

Step 1: BNP generation - generation of branch node polyhedrons (BNPs).

Step 2: BNP refinement - subdivisions of BNPs.

Step 3: Creating the tubular structure - bridging of BNPs.

Step 4: Vertex placement - reverting straightened mesh to its original pose.

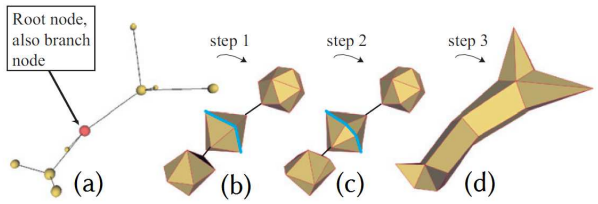


Figure 1: Steps of SQM algorithm. (a) the input skeleton; (b) generated BNPs; (c) refined BNPs; (d) BNPs bridges with quadrilateral tubes; Image from [2].

.

Straightening This is a preprocessing step of the algorithm that simplifies the generation of tubular structures. For each connection node its child is rotated, so that the edge between connection node and its child is parallel with the edge between connection node and its parent. This is useful, because during step 3 the algorithm needs to generate straight tubes only and does not need to take rotation into account.

BNP Generation A Branch Node Polyhedron (BNP) is a polyhedron assigned to a branch node. Vertices of a BNP correspond to a set of points that are generated by intersecting the sphere assigned to a branch node with each edge connected to said branch node. We will call these vertices as intersection vertices. To form a BNP intersection vertices are triangulated. After that each triangle is split into six triangles by inserting one vertex in the middle of each triangle and in the middle of each of the edges of the triangle. These vertices are then projected back onto the sphere associated with a branch node. This projection is needed because if the intersection vertices are coplanar, or nearly coplanar the generated polyhedron would have zero volume, or very small volume, respectively. The result of this step can be seen in Figure 1b.

BNP Refinement During step 3 of the algorithm, the BNPs connected via path are bridged with tubes consisting solely of quadrilaterals. This is done by connecting the one-rings of two corresponding intersection vertices with faces. To ensure that we can use only quadrilaterals the one-rings need to have the same valence. Each BNP is refined so that the valence of two intersection nodes lying on the same path are equal. We take the notion of a Link Intersection Edge (LIE): "An LIE is simply a set of edges in a subdivided BNP which belong to the links of two path vertices", from [2]. During the refinement phase only one representative edge of each LIE is subdivided. Subdivided BNPs are displayed in Figure 1c.

Creating the Tubular Structure After previous step of the algorithm, connected BNPs can be joined by a tube formed by quadrilaterals. The tube is divided into segments. Each of the segments corresponds to a connection node. Vertices corresponding to a certain connection node are projected onto its corresponding sphere. Leaf nodes are terminated with a triangle fan, which central vertex corresponds to the leaf nodes position. The result is illustrated in Figure 1d.

Vertex Placement The base mesh is now finished. All that remains is to reverse the rotations used to straighten the input skeleton. After final vertex placement the resulting mesh is smoothed with three iterations of Laplacian smoothing and attraction scheme.

Discussion Because SQM generates BNPs, it resembles the geometry of the input skeleton even without smoothing or evolution of the mesh. SQM produces small number of triangles because after the joining step triangles remain only in parts of the mesh corresponding to leaf nodes. Limitations of the algorithm are:

1. The root of the input skeleton has to be a branch node, as discussed in McDonells Skeleton Based Interactive 3D Modelling [3].
2. SQM can not generate a base mesh from linear skeletons without branching.
3. SQM supports only a sphere defined for each node of a skeleton to represent the local geometry where a more general input as ellipsoids may be desired.
4. A different termination method for leaf nodes, for example capsule termination, may be desired.
5. SQM can not handle implicitly defined cycles in the input skeleton.

The goal of our adaptation of SQM algorithm will be to improve upon all of the listed drawbacks as well as moving final vertex placement on GPU.

4 Our Base Mesh Implementation

Skeleton Straightening Skeleton straightening is a preprocessing step that simplifies bridging of branch node polyhedrons. Straightened skeleton is a skeleton which nodes in every path between two branch nodes, two leaf nodes, or a branch node and a leaf node are co-linear. In addition we have added an extra condition that angles between branch nodes child nodes should be the same in straightened skeleton as they are in the input skeleton. To achieve the first condition for each connection node, we take the normalized direction of a vector formed by connection nodes parents position and connection nodes position. The direction vector can be seen in Figure 2 as the green arrow. Then we project the child node onto the direction vector. The projected position is the position of the child node in the straightened skeleton. We then calculate rotation between connection nodes child original position and its new position, in respect to the position of connection node. Finally, we rotate all descendants of the connection node. In order to conform to the second condition, at each branch node we do not alter the position of its child nodes.

Figure 2: Skeleton straightening. Left: input skeleton; Right: straightened skeleton.

Skinning In final vertex placement, we need to revert the rotations applied to the input skeleton during straightening. We have decided that the best solution is to use skinning since it can be implemented on GPU and we wanted to move all post-processing on the GPU. Straightened skeleton represents bind pose for skinning purposes and the input skeleton represents reference pose. Now we can calculate rotations, represented as quaternions, required to transform bind pose to reference pose. Traditionally, this would require to find the rotation between two corresponding nodes in respect to their parent. Rotating all child nodes in bind skeleton using the same rotation and propagate the rotation calculation to child nodes. However, since we know precisely how bind pose was constructed, we can exploit this knowledge and avoid the rotation of child nodes. In fact, we do not even need the bind skeleton itself because the positions can be calculated from reference pose. We want to calculate the rotation that would transform a node from its bind pose to its reference pose. We know that the nodes parent is already in reference pose. We also know that bind pose was constructed in such a way that all connection nodes childes are co-linear and preserve the distances between nodes. That means from nodes parent reference pose we can calculate where would the node be in bind pose, if we would apply on it the same transformation matrices as were applied to its parent node. The distance between parent and child nodes remains constant in both poses. And the direction at which the child node would be in bind pose is the same as the direction from its grandparent node to its parent node. Now we only need to store the rotation between calculated child node position in bind pose and its actual position in reference pose with respect to its parent node. The following formula demonstrates the calculation of a quaternion required to transform one node from straightened bind pose to input reference pose:

$$
\begin{aligned}
node &\leftarrow nodeInReferencePose \\
parent &= node.Parent \\
grandParent &= parent.parent \\
distance &= dist(node, parent) \\
direction &= normalize(parent - grandParent) \\
nodeInBind &= parent + distance * direction \\
u &= normalize(nodeInBind - parent) \\
v &= normalize(node - parent) \\
rotation &= QuaternionBetweenVectors(u, v)
\end{aligned}
$$

BNP Generation We generate BNP as in original SQM algorithm. First we generate intersection vertices. Second we triangulate and subdivide these vertices. The newly inserted vertices now should be projected onto the sphere associated with their corresponding branch node. However, a detailed description of this projection was not given in the original SQM article. We have explored various possible projections. In the end we have decided to use a ray-sphere intersection. The sphere is branch nodes corresponding sphere onto which we want to project new vertices. The origin of the ray is the position of each newly inserted vertex. The direction of the ray is mean normal of the faces that are connected with the vertex. This means that for the vertices in the center of each face the normal of the subdivided face is used. For vertices inserted in the middle of each edge the mean normal of faces corresponding to that edge is used. This method does work if the center of the sphere is not in the generated BNP as well as if the generated BNP is coplanar.

BNP Refinement During BNP refinement we always split only representative edges of each LIE. In order to maintain roughly equal distribution of edges in a LIE we are applying a smoothing scheme after each subdivision. The smoothing is very important, because gener-

ated base mesh quality directly depends on the smoothing scheme. Ideally, the length of each edge in a smoothed LIE would be equal. However since smoothing is applied after every subdivision, the smoothing algorithm should be reasonably fast. We propose three smoothing schemes. These smoothing schemes are illustrated in Figure 3, where the polyhedron from Figure 3a is smoothed with various smoothing schemes.

Averaging smoothing calculates new position for each vertex on a LIE by averaging vertices in its one-ring neighbourhood. We start with the last vertex of a LIE, that is the vertex on the last edge of a LIE and move towards the first vertex. We move each vertex, except the first and the last vertices, to the barycentre of its one-ring neighbourhood and project them back onto the sphere corresponding to BNPs node. The resulting smoothed polyhedron is shown in Figure 3b. This approach is iterative and would need several iteration to achieve global optimum, however we have found that one iteration is enough for our needs.

Quaternion smoothing calculates a quaternion representing the rotation from the first vertex of each LIE to its last vertex. From each quaternion we extract its corresponding axis of rotation and angle of rotation. We smooth only points between the first and the last vertex so the calculated axis of rotation and angle are constant. During each smoothing step we first count the number of vertices in a LIE. Then we divide the angle of rotation by that number and form a new quaternion from already calculated axis of rotation and the newly calculated angle. For each vertex in a LIE between first and last we apply the rotation stored in the quaternion and update its position. This method produces LIEs that lie on small circles of their corresponding sphere. The spacing between vertices is regular and thus its very suitable for our needs. The result of quaternion smoothing is shown in Figure 3c.

Laplacian smoothing adapts the algorithm described in [1]. The weights used for smoothing are based on the one-ring area of each vertex. We use one iteration of Laplacian smoothing and then project the new vertices back onto their corresponding sphere. The smoothed mesh is shown in Figure 3d. The result is not as good as either Avaraging or Quaternion smoothing.

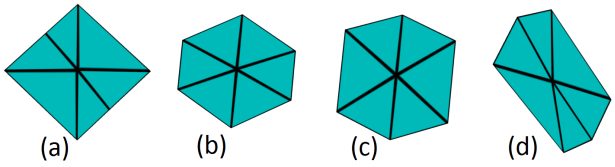


Figure 3: LIE smoothing schemes. (a) original poylehdron; (b) polyhedron after applying averaging smoothing; (c) quaternion smoothing; (d) Laplacian smoothing.

BNP Joining After refinement of BNPs intersection vertices connected via path have the same valence. Now BNPs can be joined by tubes consisting from quadrilaterals only. We loop through each branch node in a depth-first search from skeletons root. We process each BNP in the following manner. We start with the whole BNP Figure 4a. We loop through all intersection vertices corresponding to current BNP. We remove each intersection vertex and its corresponding faces and edges from current BNP. In Figure 4b we can see the removal of third intersection vertex after first and second intersection vertices were joined. After the removal of an intersection vertex we continue joining all nodes on the path that produced the removed intersection vertex. For each node, we generate new vertices and connect them with corresponding vertices from previous node. If the path leads to a branch node we remove the destination branch node corresponding intersection vertex and faces and edges connected to it. The tube generated from connection nodes is then joined with destination intersection vertex former one-ring. This approach is more suitable for our data structure than the approach proposed in SQM. Splitting a quadrilateral face into two faces is equally difficult as creating two new faces. That means the split operation would need more time in our data structure as our approach.

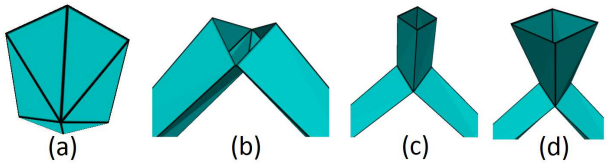


Figure 4: BNP joining process. (a) polyhedron before joining; (b) polyhedron with removed faces corresponding to an intersection vertex; (c) new vertices for connection node before projection; (d) projected vertices of connection node.

Final Vertex Placement We use quaternions calculated during skeleton straightening. For each skeletal node we accumulate the final rotation in a matrix. Matrices are used because they are more suitable for GPU calculations than quaternions. Linear blend skinning, as described in [5], is used to combine skinning matrices corresponding to each vertex on GPU. We apply skinning transformation on GPU in tessellation shaders.

5 Our Base Mesh Improvements

In this Section we propose solutions for several limitations of original SQM algorithm discussed in Section 3. We also describe how we implemented each solution.

1. Root That Is Not a Branch Node If the root of the input skeleton is not a branch node and a branch node is present in the skeleton, we can find it with a depth first search. When we have at least one branch node we can

re-root the tree so that the located branch node would be the root of the tree. This change simplifies the modelling process as user does not need to be aware of the number of neighbours of the root node.

2. Linear Skeletons Linear skeletons, which do not have branch nodes, lack the initial geometry that is generated during BNP generation step. Additional nodes could be inserted into the input skeleton to form at least one branch node, but we have found that it needlessly disturbs the flow of the output mesh. Instead we decided to use a different approach. We introduce an additional input parameter N which specifies how many vertices should be generated, for each node of the linear skeleton. This parameter does not decrease the robustness of our approach, because additional vertices are generated during tessellation and the original number of vertices is negligible.

First step of the algorithm is setting the root to be the head of the input linear skeleton. Next step of the algorithm is straightening of the input linear skeleton. The input skeleton is shown in Figure 5a. Next, N vertices are generated around first connection node, which is a child of the root node. These vertices are distributed regularly around the node by slerping a quaternion, which center of rotation is nodes position, axis of rotation is the direction from connection node to root node and magnitude is $360/N$. Newly generated vertices are then joined with other vertices as in original base mesh algorithm. Leaf nodes form a triangle fan and connection nodes form a tube of quadrilaterals. The joined linear base mesh is shown in Figure 5b. Skinning matrices are used to transform the generated linear skeleton into its input pose Figure 5c.

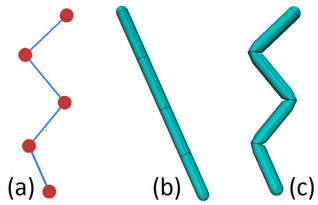


Figure 5: Linear base mesh generation. (a) input linear skeleton; (b) straightened and joined linear skeleton; (c) final linear base mesh.

3. Ellipsoid Nodes An ellipsoid can be defined as a sphere with associated transformation matrix. We take advantage of this representation of ellipsoids. Instead of more complex ray-ellipsoid intersection that would have to be computed at each ellipsoid node, we have decided to represent each ellipsoid node as a sphere and a transformation matrix. First our base mesh algorithm is evaluated as described in Section 3 with spherical nodes. After that we send the transformation matrices corresponding to each ellipsoid node to GPU. The vertices corresponding to each ellipsoid node are transformed directly in vertex shader. Thanks to this, ellipsoid nodes require minimal extra computing resources from CPU. The results can be seen in Figure 6.

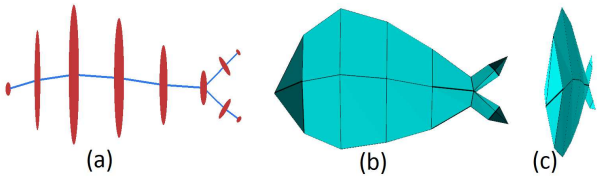


Figure 6: Ellipsoid nodes. (a) skeleton with ellipsoid nodes specified; (b) base mesh generated from skeleton; (c) base mesh from different angle.

4. Capsule Ending A capsule is a hemisphere generated at each leaf node of the input skeleton. Generation of capsules can be approached in two ways. The first is to generate a capsule at each leaf node corresponding to its radius. The second is inserting additional nodes into the input skeleton with decreasing radius that would approximate a capsule. We have implemented the second approach because it fits nicely into our pipeline. Capsules generated this way, can be directly tessellated on the GPU without any additional processing. At each capsule leaf node, we insert additional nodes into the input skeleton, proportional to the radius of the capsule node. The radius of each node is decreased according to the following equation: $newRadius = \sqrt{nodeRadius^2 * (1 - step^2)}$ where $nodeRadius$ is the radius of capsule node and $step$ is a number between $(0 - 1]$, that represents the distance from center of the capsule to its edge. Final tessellated capsule is shown in Figure 7.

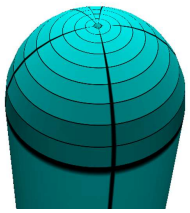
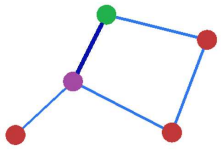


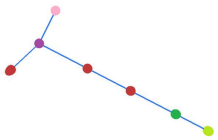
Figure 7: Capsule generated by our algorithm.

5. Cyclic Skeletons Our last improvement is generation of base meshes from cyclic skeletons. The cycle can be located anywhere in the input skeleton. The base algorithm could not be modified to allow generation of cyclic meshes, because during BNP refinement step of the algorithm a cycle could cause an infinite loop. However we can modify the input skeleton in a way that would allow us to generate cyclic skeletons. As the input we have a cyclic skeleton Figure 8a. Cyclic edge is marked with dark blue color and cyclic nodes with dark green (upper node) and dark violet (lower node) colors. First, we split the cycle

by removing the cyclic edge. To each cyclic node we add an extra child node as shown in Figure 8b. Light green node for dark green cyclic node and light pink node for dark violet cyclic node. These new nodes serve to preserve the skinning matrices that will rotate tubes generated from cyclic nodes to face each other. This can be seen in Figure 8c. Base mesh was generated as described in Section 4 with one exception. We do not generate geometry for light green and light pink nodes. Now the gap between cyclic nodes should be closed. We first project vertices associated to each cyclic node to a plane with origin at $O(0,0,0)$ and normal $n(0,1,0)$. Next, we normalize the vertices so that vertices associated with violet node lie at a circumference with radius 1 and vertices associated with green node lie at circumference with radius 2. The position of projected points is shown in Figure 8e, outer points correspond to dark green node and inner points correspond to dark violet node. Now we execute a Delaunay triangulation on the transformed points. After the triangulation is done, we exclude triangles generated solely between inner or outer vertices. The remaining triangles, Figure 8f represent the faces that should be generated between vertices of cyclic nodes in our generated base mesh in order to close the gap. Final cyclic mesh is shown in Figure 8d.

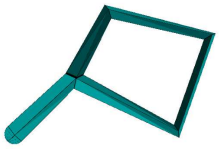


(a) Cyclic skeleton, cyclic edge marked with dark blue color, cyclic nodes with dark green and dark violet

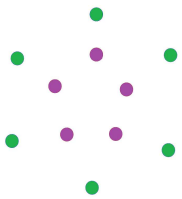


(b) Split cycle with one inserted node for each former cyclic nodes light green for green node and light pink for violet node

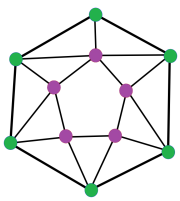
(c) Generated base mesh before the cycle is closed



(d) Generated base mesh after the cycle is closed



(e) Vertices corresponding to green and violet cyclic nodes projected onto the same plane



(f) Faces generated by triangulation that will be used between original unprojected vertices

Figure 8: Cyclic skeleton base mesh generation.

6. Tessellation Tessellation shaders available since OpenGL 4.0 are used to tessellate the generated base mesh. Two connected spherical nodes, a parent and a child, implicitly define a truncated cone between them. The base of the cone has the radius of parent spherical node and the top of the truncated cone has the radius of child spherical node. Each vertex generated during tessellation is projected onto this cone. The projection is done by translating the vertex along its normal until it reaches the surfaces of the cone. A generated base mesh is shown in Figure 9a and after tessellation in Figure 9b. However during this step the generated base mesh gains volume and the newly generated vertices can intersect the tessellated base mesh. This effect can be seen in Figure 9c. To recover from this situation, we detect sharp vertices in the input mesh and apply a radius scaling scheme. Sharp vertices are vertices which faces are forming acute angles. In tessellation shader we have access only to one patch and its vertices. So we compute the sharpness of each vertex by comparing and thresholding the normal of each vertex with the direction of the patch. The smaller the angle between vertex normal and patch direction is, the sharper the vertex is. We apply Bézier curves to modify the radius of the truncated cone. We use Bézier curves that yield values between $[0,1]$. For each tessellated vertex its *distance* from the beginning of the patch is calculated. The distance is equal to tessellation parameter v computed by the GPU. Scaling reduction factor is calculated by sampling a point on Bézier curve at point $t = distance$. The radius of each vertex is then multiplied with calculated factor. The smoothed mesh is shown in Figure 9d. Currently, the scaling bezier curve is constant, but it could be dynamically changed based on the sharpness of the vertices.

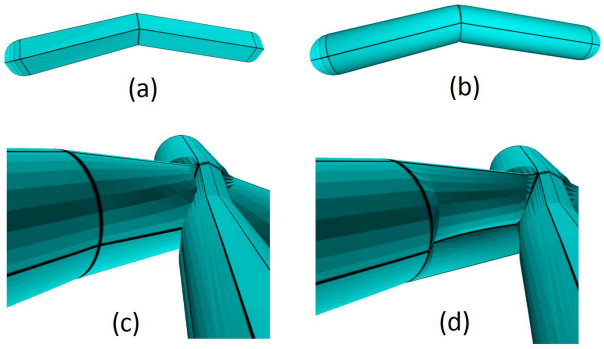


Figure 9: Tessellation. (a) non-tessellated mesh; (b) tessellated mesh with 20 subdivisions; (c) tessellated mesh with self intersection; (d) tessellated mesh with scaling.

Model	Node Distribution				Timing of steps in milliseconds				
	#nodes	#branch	#connection	#leaf	straightening	generation	subdivision	joining	**Total**
worm	23	21	2	0	0	0	0	5	5
dummy	56	2	49	5	0	4	2	15	21
cycle	14	1	12	1	0	4	2	18	24
octopus	131	1	117	13	1	9	5	54	69
dummy 2	140	5	122	13	1	11	6	39	57
goat	150	9	123	18	1	16	10	47	74

Table 1: Table showing statics of base mesh algorithm. From left to right: name of the model, node distribution in skeletal structure, timing of each step of the algorithm measured in milliseconds.

6 Results

The algorithm was implemented in C++ in Visual Studio 2012. Mesh is stored in open source half-edge data structure OpenMesh 2.2. OpenGL 4.3 is used to visualize the algorithm and tessellate the generated base mesh. We have also developed an interactive system, where the user can create, save and load skeletons. Nodes can be edited directly with mouse input, or using node property inspector. New nodes can be inserted into the skeleton with mouse clicks.

Performance of the algorithm is shown in Table 1. The table shows from left to right: the name of measured model, distribution of branch, connection and leaf nodes in the model and time required for each step of the algorithm measured in milliseconds. Time was measured on Intel® Core™ i7-3615QM a four core processor with each core clocked at 2.3 GHz. From the table, we can see that the joining step, during which the tubular structure is generated, took the most time. Therefore, it is a candidate for optimization since other steps of the algorithm took nearly no time to execute. However, even at current speed we can generate base meshes at interactive frame rate.

Our algorithm is also capable of generating base meshes from skeletons on which SQM would fail. For example, in Figure 10a we can see a fish produced in SQM without ellipsoid nodes. The generated base mesh resembles an eel. Our algorithm with ellipsoid nodes, Figure 10b, produces a base mesh that corresponds to a fish. In Figure 10d we can see a cyclic mesh generated by our algorithm. We tried to generate similar mesh in SQM from the same skeleton, Figure 10c. Producing a similar mesh in SQM is not possible as the cycles do not lie in symmetrical region of the input skeleton and SQM would not close them.

7 Conclusion

We have managed to improve all the drawbacks discussed in Section 2. Our algorithm is capable of generating base meshes from linear skeletons, explicitly defined cyclic skeletons, as well as from skeletons with root that is not a branch node. We have moved Final Vertex Placement step of the algorithm on GPU. Lastly, we can set arbitrary ellipsoids at each skeletal node and improve visual quality of generated base mesh in tessellation shaders.

References

[1] Oscar Kin-Chung Au, Chiew-Lan Tai, Hung-Kuo Chu, Daniel Cohen-Or, and Tong-Yee Lee. Skeleton extraction by mesh contraction. *ACM SIGGRAPH 2008 papers*, pages 1–10, 2008.

[2] J. A. Bærentzen, M. K. Misztal, and K. Welnicka. Converting skeletal structures to quad dominant meshes. *Computers & Graphics*, 36(5):555–561, 2012.

[3] Michael Mc Donnell. Skeleton-based and interactive 3d modeling. Master's thesis, Technical University of Denmark, 2012.

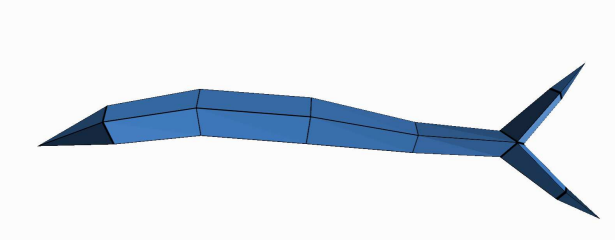
[4] Zhongping Ji, Ligang Liu, and Yigang Wang. B-mesh: A fast modeling system for base meshes of 3d articulated shapes. *ComputGraphForum*, 29(7):2169–77, 2010.

[5] Ladislav Kavan, Steven Collins, Jiri Zara, and Carol O'Sullivan. Skinning with dual quaternions. In *2007 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 39–46. ACM Press, April/May 2007.

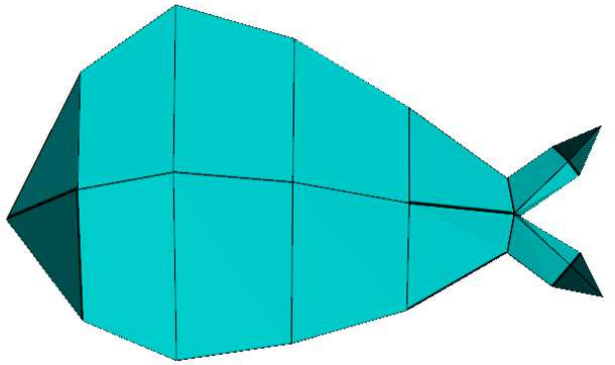
[6] Luc Leblanc, Jocelyn Houle, and Pierre Poulin. Modeling with blocks. *Vis Comput*, pages 555–563, 2011.

[7] Vinod Srinivasan, Esan Mandal, and Ergun Akleman. Solidifying wireframes. *Proceedings of the 2004 bridges conference on mathematical connections in art, music, and science*, 2005.

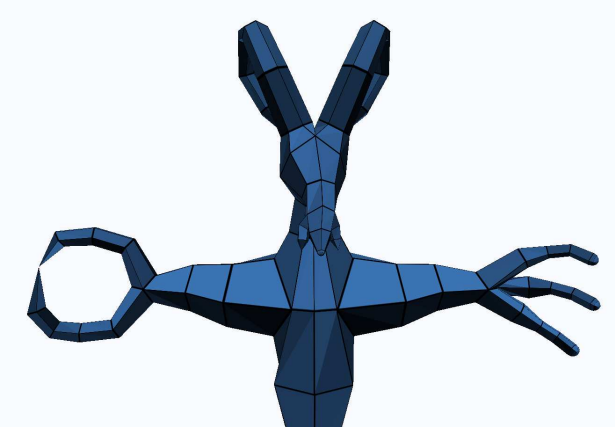
[8] Roger Tam and Wolfgang Heidrich. Shape simplification based on the medial axis transform. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 63–71, Washington, DC, USA, 2003. IEEE Computer Society.



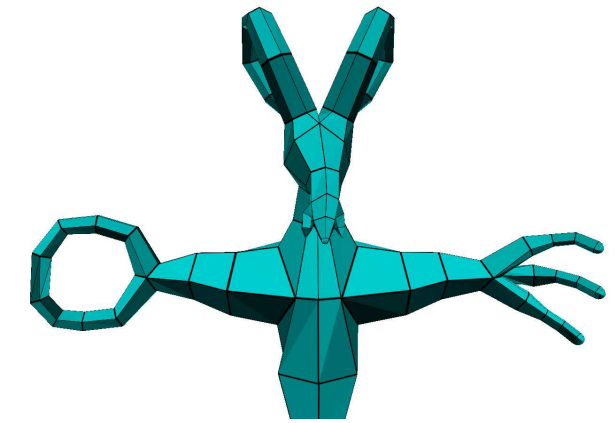
(a) Fish without ellipsoid nodes as would be generated by SQM



(b) Fish with ellipsoid nodes generated by our algorithm



(c) Mesh from cyclic skeleton as would be generated by SQM



(d) Mesh from the same skeleton generated by our algorithm

Figure 10: Comparision of SQM to our implementation.

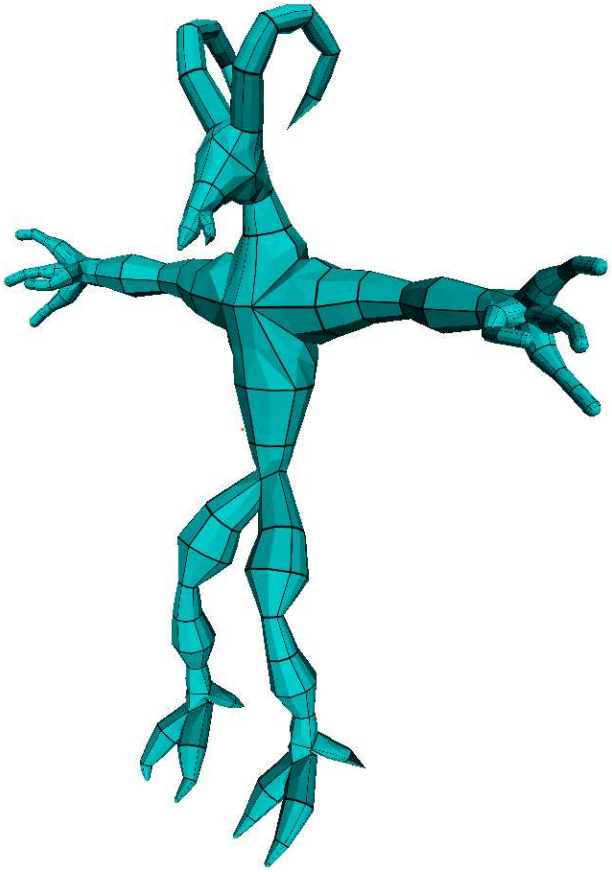


Figure 11: Goat creature generated with our base mesh algorithm.

Applying Engineering Constraints in Digital Shape Reconstruction

István Kovács*
Supervised by: Tamás Várady†

Budapest University of Technology and Economics

Abstract

The goal of digital shape reconstruction is to create computer models from point clouds; however, inaccuracies may occur due to the noise of measured data and the numerical nature of the algorithms used for fitting. As a consequence, faces will not be precisely parallel or orthogonal, smooth connections will be of poor quality, axes of concentric cylinders may be slightly tilted, and so on. In this paper we present algorithms to eliminate these inaccuracies and create perfect models, which are suitable for downstream CAD/CAM applications. We extend a formerly published technology [1] in two areas. We propose methods to (i) automatically set up hypotheses for likely geometric constraints and (ii) compute global constraints related to the whole object, such as, an optimal coordinate system and associated grid, or the best - full or partial - axes of symmetries. In this paper we investigate planar contours with constraints; nevertheless, extending this technology to 3D is in progress, as well. A few interesting examples will be presented to show how constrained fitting can improve the quality of reconstructed objects.

Keywords: Reverse engineering, Constrained fitting, Symmetry detection

1 Introduction

Digital shape reconstruction (reverse engineering) is an expanding, challenging area of Computer Aided Geometric Design [12]. This technology is utilized in various applications where a given physical object is scanned in 3D, and a computer representation is needed in order to perform various computations. A wide range of applications emerges in engineering, medical sciences, and to preserve the cultural heritage of mankind [6]. Examples include redesigning and re-manufacturing old mechanical parts, creating surface geometries from clay models, or producing surfaces matching human body parts for hearing aids, dentures, prosthetics, etc.

*kovacsi@math.bme.hu
†varady@iit.bme.hu

1.1 Digital shape reconstruction

Digital shape reconstruction consists of the following technical phases: (1) 3D data acquisition (scanning), (2) filtering and merging point clouds, (3) creating triangular meshes, (4) simplifying and repairing meshes, (5) segmentation (partitioning into disjoint regions), (6) region classification, (7) fitting surfaces (i.e. approximating the data points), (8) fitting connecting surfaces (e.g. fillets), *(9) perfecting surfaces* (including constrained fitting and surface fairing), (10) exporting to CAD–CAM systems for downstream applications.

Assume segmentation has taken place, and classification produced a surface type for each region that will best approximate the related data points. The conventional approach is to fit surfaces individually. Let us denote the surfaces by $\{s_i\}$, and the corresponding point clouds by $\{p_{ij}\}$. Our goal is to minimize the average square distances between the surfaces and the point clouds. Let $\mathbf{x}$ contain the parameters of the surfaces. Then the problem is

$$f_i(\mathbf{x}) = \sum_j d(p_{ij}, s_i)^2, \qquad f_i(\mathbf{x}) \to \min.$$

Fitting simple surfaces is generally based on solving eigenvalue problems [2] [11]; for more complex surfaces efficient numerical methods exist [9]. Fitting surfaces *separately* is likely to produce inaccuracies; fortunately, the model quality can be perfected, if we recognize and enforce various geometric constraints and then fit groups of related surfaces *simultaneously*.

1.2 Constrained fitting

Geometric constraints define relationships amongst various entities. This is a key issue in engineering design; orthogonality, parallelism, tangency, symmetry, etc. can be best prescribed by means of constraints, which are expressed in the form of various algebraic equations.

We may distinguish between constraints that has *local* effect related to pairs of curves and surfaces, such as, lines, circles, planes, cylinders, cones, extruded and rotational surfaces, and more complex constraints that *globally* determine groups of surfaces. The most frequent local constraints include

- orthogonal/parallel curves and surfaces,

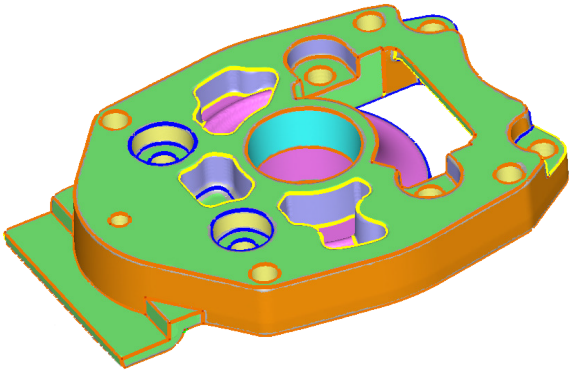


Figure 1: An engineering object with many self-contained constraints.

- concentric curves and surfaces,
- tangential curves and surfaces,
- rounded numerical values,
- fixed numerical values.

The most frequent global constraints include

- common direction for extrusions,
- common rotational axes,
- global grid,
- global axis of symmetry,
- global rotational symmetry.

The scanned data – in itself – do not carry information about the structure of the object and the constraints between its high-level geometric entities. These need to be set either explicitly by the user, or recognized and set by some "intelligent" algorithm. After individual fitting — due to noise and numerical inaccuracies — constraints will be satisfied only within some tolerances; an example with inaccurate values is shown in Figure 2. If we set a constraint system, we can enhance the model and *refit* the surfaces accordingly. This process is called *constrained fitting*.

Our goal is to minimize the average square distances between the point clouds and the surfaces while constraint equations are enforced.

1.3 Previous work

Numerical methods to solve this problem have been published earlier [14] [8]. An important paper on constrained fitting was published by Langbein et al. [3], where partial symmetries on point sets are detected based on an algebraic concept. In the paper of Mitra et al. [7] it is shown how partial global symmetries on 3D models by feature points can be detected. Our paper expands a numerical technique originally suggested by Benkő et al. [1], that can handle under- and over-constrained systems with priorities, applying a special extension of Newton's method. An interesting approach was recently published in [4], that discovers certain primitives, such as, planes, cylinders, cones and spheres using RANSAC method [10] and then sequentially enforces constraints amongst them.

1.4 Outline

In this paper we focus on algorithms, that substitute user driven, manual constrained fitting by automatic techniques. After presenting the basic algorithm of Benkő et al. [1] in Section 2, we present how to detect and enforce various hypotheses for likely *local* geometric constraints in Section 3. Then we continue with methods to detect *global* constraints, including best fit grids and optimal axes of symmetries - see Section 4 and 5, respectively. Finally, results will be illustrated by a few examples using 2D point sets and related constraints for planar curves.

2 Constrained fitting – basics

2.1 A simple example

Consider the profile curve in Figure 2(b) [13]. If we fitted these circles independently, the tangential constraints would not be satisfied, however, constrained fitting provides an appropriate solution. In this example, let c_i denote the circles to be constrained with parameters (A_i, B_i, C_i, D_i) , and the corresponding equations are $A_i(x^2+y^2)+B_ix+C_iy+D_i=0$. The average squared distance to be minimized is

$$f(\mathbf{x}) = \sum_{i,j} d_{i,j}^2 = \sum_i \frac{1}{n_i} \sum_j (A_i(x_{ij}^2+y_{ij}^2)+B_ix_{ij}+C_iy_{ij}+D_i)^2.$$

The constraint system includes

- normalization constraints $(B_i^2 + C_i^2 - 4A_iD_i = 1)$; these assure that the distances from the points closely approximate the Euclidean distances, and
- tangential constraints $(2A_jD_i + 2A_iD_j - B_iB_j - C_iC_j \pm 1 = 0)$; these assure that each pair of adjacent circular arcs shares a common endpoint and tangent.

2.2 The numerical method

Let us continue with presenting the method of Benkő et al. [1], this is necessary to understand the rest of the paper. We use the previous notations, where $d(p_{ij}, s_i)$ denotes the distance between surface s_i and point p_{ij}. Let α_i be the positive weights assigned to the i-th surface. Let $\mathbf{x}$ contain the parameters of all surfaces, and let us define the

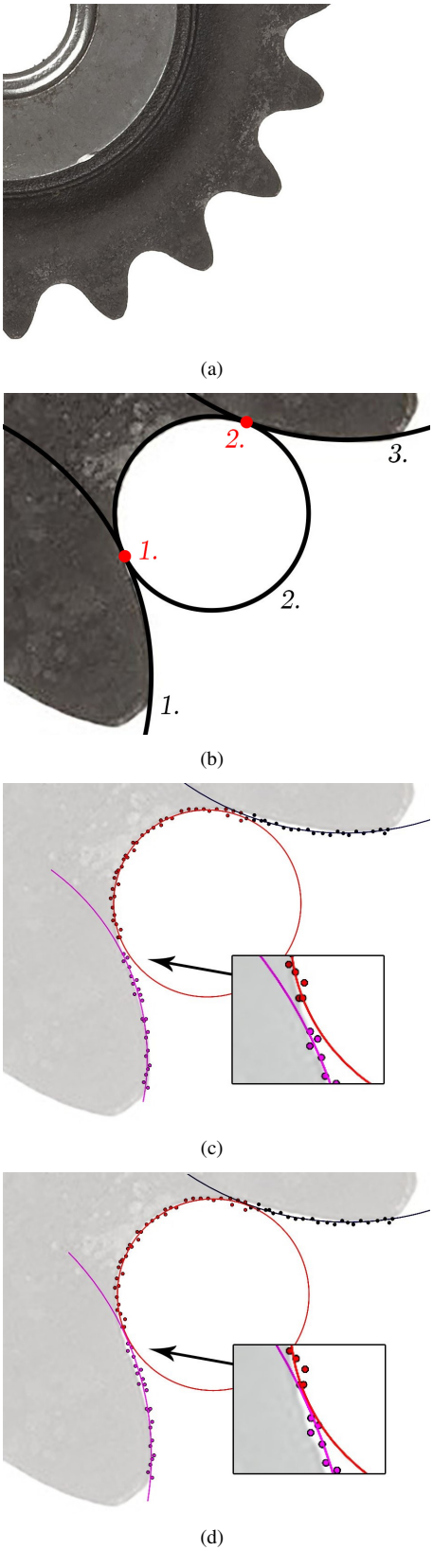


Figure 2: Constrained fitting: (a) profile of a gear wheel; (b) three circles with prescribed tangency; (c) independent fitting yields discontinuity; (d) fitting with constraints guarantees smooth connections.

constraint equations in the form of $c_k = 0$. Then we can write the global system of equations as

$$\mathbf{c}(\mathbf{x}) = 0. \tag{1}$$

We minimize the average square distance while the constraints are satisfied:

$$f(\mathbf{x}) = \sum_i \alpha_i \sum_j d(p_{ij}, s_i)^2 \tag{2}$$

Let $\mathbf{c}(\mathbf{x}) = (c_1(\mathbf{x}), ..., c_k(\mathbf{x}))$, where the constraints are ordered by priority and suppose that $f(\mathbf{x})$ and $\mathbf{c}(\mathbf{x})$ are smooth enough (at least C^2). Here we have a highly nonlinear system of equations, that we are going to solve using a special Newton iteration. We approximate $\mathbf{c}$ in first order, and f in second order. The Taylor approximations of $\mathbf{c}$ and f around $\mathbf{x}_0$ are the following:

$$\mathbf{c}(\mathbf{x}_0 + \mathbf{d}) \approx \mathbf{c}(\mathbf{x}_0) + \mathbf{c}'(\mathbf{x}_0)\mathbf{d} \tag{3}$$

$$f(\mathbf{x}_0 + \mathbf{d}) \approx f(\mathbf{x}_0) + f'(\mathbf{x}_0)\mathbf{d} + \frac{1}{2}\mathbf{d}^T f''(\mathbf{x}_0)\mathbf{d} \tag{4}$$

In each step we want to determine a small difference vector $\mathbf{d}$. Using the above equations, the problem can be written locally in the form

$$C\tilde{\mathbf{d}} = 0 \tag{5}$$

$$\tilde{\mathbf{d}}^T A \tilde{\mathbf{d}} \to \min, \tag{6}$$

where $\tilde{\mathbf{d}} = (d_1, ..., d_n, 1)$, $C = [\mathbf{c}'(\mathbf{x}_0)|\mathbf{c}(\mathbf{x}_0)]$ and A is an $(n+1) \times (n+1)$ size matrix, as follows:

$$A = \left[\begin{array}{c|c} f''(\mathbf{x}_0) & f'(\mathbf{x}_0) \\ \hline f'(\mathbf{x}_0)^T & 0 \end{array}\right]. \tag{7}$$

In order to calculate $\tilde{\mathbf{d}}$ we have to reduce it to a lower dimensional vector $\mathbf{d}^*$ by (5), such that $\mathbf{d}^*$ has only *independent* coordinates. We calculate a matrix M, such that $\mathbf{d} = M\mathbf{d}^*$, and $CM = 0$. Now the dimension of $\mathbf{d}^*$ gives us, how many independent variables exist in the system. Finally, we can solve $\mathbf{d}^{*T} A^* \mathbf{d}^* \to \min$ without constraints, where $A^* = M^T A M$, and this can be solved as a simple system of linear equations. We note that, this minimization is always solvable, the proof is based on the fact, that $f''(\mathbf{x}_0)$ is symmetric positive definite in this case.

The way of calculating M is very similar to Gauss elimination. During elimination, we can check if some of the constraints contradict to each other, or if the system is over-determined (see details in the original paper [1]).

2.3 Auxiliary objects

The use of the so-called *auxiliary objects* is an important idea in constrained fitting. We illustrate this through a simple example. Take three lines that are supposed to meet in a common point (see Figure 4(a)). We can formulate the related constraints by taking line 1 and 2, compute their intersection and constrain this intersection point to lie on

line 3; then we take line 2 and 3, and line 3 and 1 with similar constraints. This set of equations defines a relatively simple problem in a very complicated way.

An alternative solution is to introduce an auxiliary point p. This is also an unknown entity, but now we can define our constraints by three simple equations: i.e. all three lines must pass through p. Clearly, we have increased the number of unknowns in the parameter vector $\mathbf{x}$, but the system of equations – and all related Taylor approximants – have become much simpler. Note, the unknown surface parameters are generally associated with a corresponding point sets, but for auxiliary objects such a data point has no meaning. Typical auxiliary entities include a point, a point and a normal, a distance, etc.; their exclusive role is to simplify the system of equations and thus our computations.

3 Automatic detection of local constraints

Let us start with a simple example. We wonder whether pairs of lines are perpendicular or not, and we wish to incorporate additional constraints into our system, if the likelihood of being perpendicular is high. This can clearly be controlled by a user defined angular tolerance, and extra constraints will be added automatically, if two lines span an angle between $90 \pm \varepsilon$.

Formally: let $c(\mathbf{x}) = 0$ a simple constraint between two objects. Let ε be a tolerance level. The c constraint is within tolerance if and only if $|c(\mathbf{x})| < \varepsilon$, and we want to validate whether the constraint holds. For this, we introduce the following function:

$$s_\varepsilon(x) := \begin{cases} x & \text{if } |x| < \varepsilon \\ 0 & \text{otherwise.} \end{cases}$$

We observe that, if $c(\mathbf{x})$ is out of tolerance, then $s_\varepsilon(c(\mathbf{x}))$ vanishes, and the constant zero constraint will not modify our system, otherwise $s_\varepsilon(c(\mathbf{x}))$ reproduces $c(\mathbf{x})$.

A necessary condition for c is, that $c(\mathbf{x})$ represents a so-called *faithful* representation for the distance used for a given entity. (Faithful means that the true Euclidian distance or a close approximation is computed in the vicinity of the curve/surface to be fitted, see details [1]). For example, for the *line meets point* constraint, we must use a normalized line-point distance function

$$c_{pl}(\mathbf{x}) = \frac{|Ax_0 + By_0 + C|}{\sqrt{A^2 + B^2}}.$$

Also note that s_ε is not continuous, but a piecewise continuous function, so if we calculate the derivative numerically, we need to make it piecewise, as well.

To detect constraints automatically, we take the modified constraints for all object pairs. The numerical method will enable constraints only that are within the related tolerance level. The user typically defines different tolerances for different – parallel, perpendicular, tangential,

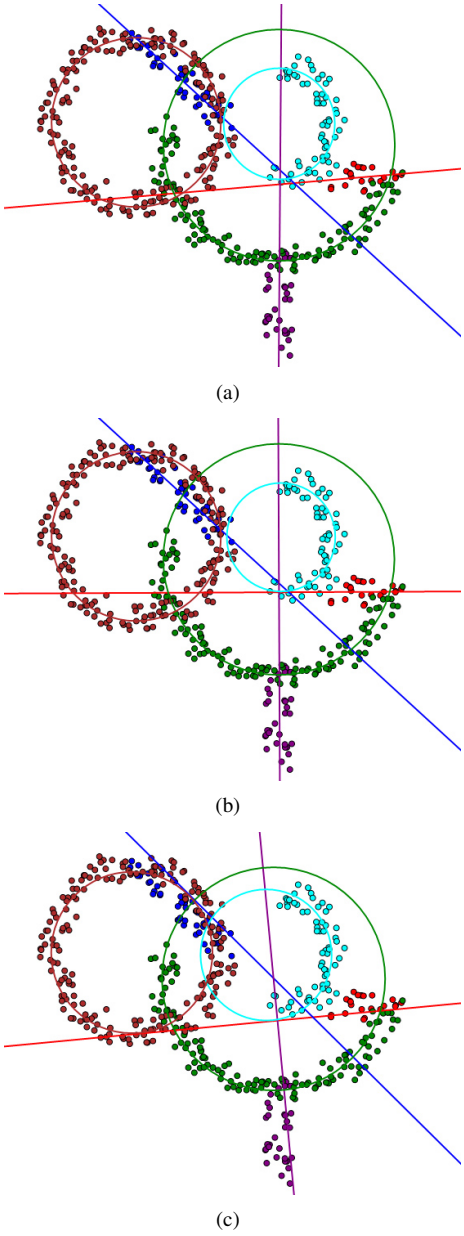


Figure 3: Automatically detected constraints: (a) — initial state; (b) and (c) — different configurations created by different tolerance levels

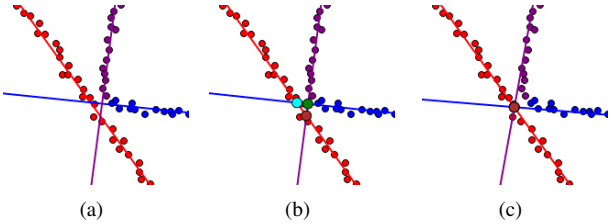


Figure 4: Three lines meet at a common point: (a) initial state; (b) pairwise intersection (auxiliary points); and (c) enforce the 'three points are equal' constraint.

concentric, etc. – constraints. Let us denote the set of objects by $S = \{s_i\}$, and the constraint types by $\{c_j\}$, such that $c_j(s_1, s_2)$ denotes an actual constraint between s_1 and s_2. Then for all c_j, consider the following constraint set:

$$C_j = \{s_{\varepsilon_j}(c_j(s_1, s_2)) | s_1, s_2 \text{ suitable for } c_j\}.$$

Thus the global constraint system includes the explicitly defined constraints, and the C_j-s, i.e. the 'likely' constraint set.

A somewhat artificial example with three circles and three lines can be seen in Figure 3 that shows different configurations created by different constraint tolerances. Compare cases (b) and (c). The angular tolerances of 'lines orthogonal' are (b):10, (c):10 degrees. The distance tolerances of 'line passes through the center of a circle' are (b):10, (c):10 units, and 'line is tangent to circle' are (b):15, (c):25 units, respectively.

We can also handle more complex local (i.e. not pairwise) constraints. For example, take the previously mentioned *three lines meet in a single point* constraint in Figure 4. We may create auxiliary intersection points for all three pairs of lines, and by means of a corresponding *line close to point* constraints the algorithm can detect, whether the three intersection points are "likely to be" coincident or not. In the former case the three lines will be fitted simultaneously, enforcing a common point of intersection.

4 The best fit global grid

In this section, we investigate how to detect and create a best fit *'grid'* object and set the corresponding constraints.

The grid is represented as a 5 dimensional auxiliary object with the following parameters:

- the orientation of the grid (n),
- the origin of the grid (p_0),
- and a positive constant, the width of the cells (d).

Note that, the above parameters are not uniquely defined, since we can select all intersection points of the grid as origin, and we have four ways to define the orientation.

We can define constraints for the grid in a similar way, as earlier. For example, the constraint of a line ($Ax+By+C=0$) is orthogonal/parallel to the grid can be given as $c(\mathbf{x}) = \min(|An_1 - Bn_2|, |An_2 + Bn_1|) = 0$. We can define the point meets grid constraint as $\langle p - p_0, n\rangle / d$ and $\langle p - p_0, n^{\perp}\rangle / d$ are integers. So the most important constraints are the following:

- certain parameters (n, p_0, d) are fixed,
- points are contained in the grid,
- lines are orthogonal/parallel to the grid,
- lines lie on the grid lines.

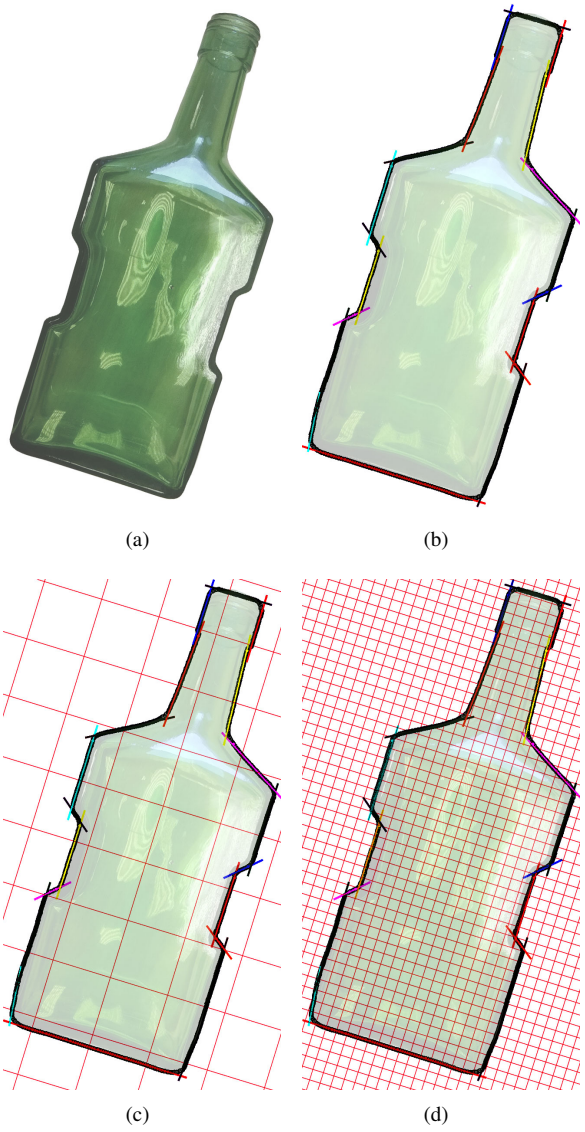


Figure 5: Detect grid: (a) original glass object; (b) segmented profile with straight segments; (c) optimal orientation; (d) final fit with optimal cell size.

For detecting the above constraints, we can use the automatic methods shown earlier, but this will work only, if the grid has been initialized 'almost perfectly'. Alternatively, we suggest an algorithm based on the following four basic steps (a related example can be depicted in Figure 5):

1. determine the best orientation,
2. fit the corresponding lines,
3. determine the best width parameter,
4. refit the objects matching the enhanced grid.

4.1 Determine the best orientation

Assume that, each line belongs to a straight section. Let us denote the length of l_i as $h(l_i)$, and the angle from x-

axis of its normal vector l_i in radian as $\angle(l_i)$. With respect to the grid, α and $\alpha+\pi/2$ have the same orientation, so we work with angles modulo $\pi/2$. The solution is given by clustering the angular values. We associate a radius with each cluster, that depends on a tolerance level and a weight by $w(S)=\sum_{l\in S}h(l)$. We select the best cluster (where $w(S)$ is maximal), and the weighted average of the angles will yield the best orientation.

4.2 Determine the best cell size

After setting up the best oriented grid, we fit the corresponding lines by the automatic method presented in Section 3. The next problem is determine the best common divisor of the distances between the parallel lines.

Let us denote the lines fitted according to the optimal orientation by $\{l_i\}$, and the absolute distances between the parallel lines by $\{d_{ij}\}=\{n_l\}$. If d is a suitable width of the grid cells, then the average remainder by $\{n_l\}$ is small. The sum of remainders can be written in the form

$$\delta(d)=\sum_l \min\left(\left\{\frac{n_l}{d}\right\},1-\left\{\frac{n_l}{d}\right\}\right),$$

where $\{x\}$ denotes the fraction part of x.

Now the goal is to find the minimum of $\delta(d)$ in the $[d_{min},d_{max}]$ interval. It is easy to see, that the function δ is piecewise monotone, and the monotonicity drops at numbers being in the form of n_l/k, for certain n_l-s, where k is a positive integer. Therefore, we need to search for the minimum only at these points, and we can find this in $O(N^2 n_{\max}/d_{min})$ steps, where N is the number of distances, and $n_{\max}=\max_l n_l$.

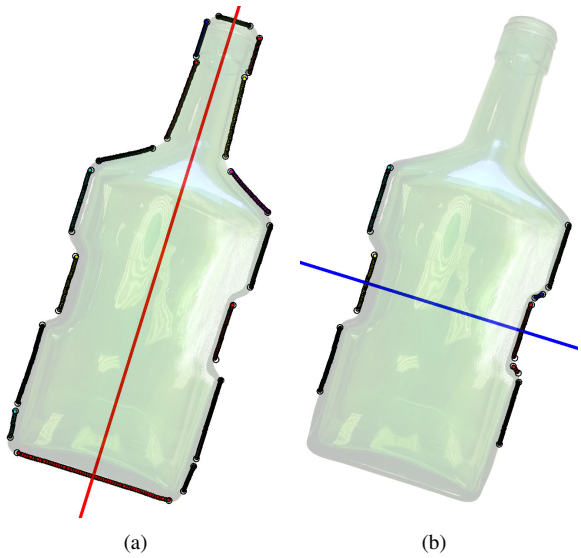
After setting up the optimal grid size, we can automatically detect the lines that satisfy all the grid constraints.

5 Estimate global symmetries

The second area of setting global constraints is the computation of axes of symmetry. We investigate algorithms for curves consisting of straight segments and circular arcs. First we determine all potential axes that may occur, then evaluate and prioritize them, and finally select the best one(s). Let $P=\{p_i\}$ denote the endpoints of the segments and the centers of the circles, and $L=\{l_i\}$ the lines.

The main steps of the algorithm are the following (see also Figure 6):

1. Collect all perpendicular bisectors between the points of P, and all angular bisectors between the lines of L. These bisector lines are called *auxiliary lines*.
2. Determine clusters of the auxiliary lines.
3. Evaluate the clusters (i.e. compute the corresponding axes and evaluate their 'measure' of symmetry).
4. Select the best axis (axes), and apply constrained fitting accordingly.



(a) (b)

Figure 6: Detected axis of symmetry: (a) the best axis of symmetry (88.2%); (b) the second best axis of symmetry (35.9%).

The set of *auxiliary lines* A contains the perpendicular bisectors between the points of P: $A_1=\{\text{PBisector}(p_i,p_j):i<j\}$, and the angular bisectors between the lines of L: $A_2=\{\text{ABisector}(l_i,l_j):i<j\}$. Now we cluster these in two steps. First by the argument of the normal vectors (modulo π), then by the distances from the origin. For each cluster C, let l_C denote the average of C, these are the *axis candidates*. The number of elements in a given cluster is not necessarily the best quantity to measure the extent of symmetry; it is better to locate the corresponding symmetric parts by the related axis candidate and compute their arc lengths. With other words, symmetries amongst many small segments will be considered less important than those of a few large segments.

The clusters also provide information about symmetries of circular arcs, which help to enhance these computations. For each pair of arcs, we determine the corresponding arcs, or parts of arcs, that can really be considered as symmetric. The sum of these arcs yield additional weights to qualify the axis candidates.

We generally define constraints for axis of symmetry using auxiliary objects, as well. For example,

- *an axis is a perpendicular bisector of a segment*,
- *axis is an angular bisector of two lines*, etc.

Finally we perform constrained fitting according to the best axis of symmetry.

6 Examples

In this section, we present some examples using our algorithms of constrained fitting.

6.1 Case Study 1 - Gear wheel

As it was shown earlier in Figure 2, if we fit three circles independently, there will be small gaps between the adjacent arcs without tangential continuity, thus yielding a profile curve with poor quality. To avoid this, we must apply constrained fitting, as shown in Figure 2(d). In this case the least-squares deviation will be somewhat worse, but the prescribed constraints will be satisfied. This is illustrated numerically in Figure 7. The average least-squares errors increased by at most 15.5%, but this is still negligible compared to the magnitude of the circle radii. The radius of the first circle has been increased by almost 20%, actually this is the correct value. The reason for this is that the points lie on a relatively small segment of a large circle. In these cases the computed value of the best-fit radius is not so robust, and this may lead to relatively large changes once the constraints are enforced.

	Independent fit	Constrained fit	Deviation
1. error	1.015	1.201	15.50%
1. radius	125.762	155.810	19.28%
2. error	1.138	1.216	6.44%
2. radius	50.590	51.683	2.11%
3. error	0.822	0.855	3.77%
3. radius	145.962	156.278	6.60%
1. constraint	2.142	0	
2. constraint	1.010	0	

Figure 7: Numerical analysis of unconstrained vs. constrained fitting for the profile curve in Figure 2. (i) Least squares errors and (ii) radii of the three fitted circular arcs, (iii) estimated deviation errors at the connecting points of the circles.

6.2 Case Study 2 - Bottle

To demonstrate our algorithms for detecting global constraints we used the profile curve of a glass. Both algorithms (detecting grid and axis of symmetry) have produced satisfactory results.

Grid detection has already been shown in Figure 5; the algorithm located the best orientation and cell size for the grid, when performed constrained fitting. As explained earlier, the algorithm has a dynamic behaviour, i.e. it adjusts, which constraints are actually taken into consideration when the system of equations is finally solved. Close views of Figure 5 are shown in Figure 8. We can see the middle purple line in 8(b) is almost parallel to the grid, but the angle is out of the tolerance level. The same thing happens on the 8(e) with the first yellow line. Another interesting effect, is the third purple line in 8(e), which is fitted to the orientation, but not fitted to the grid. This illustrates that, the algorithm is sensitive to the local inconsistencies, and adaptively determines the constraints for the optimal global grid.

Symmetry detection was demonstrated in Figure 6. The algorithm determined two axes of symmetry with symmetry levels 88.2% and 35.9%, respectively. The first axis is the global axis of symmetry, while the other one indicates a local symmetry in the middle of the object.

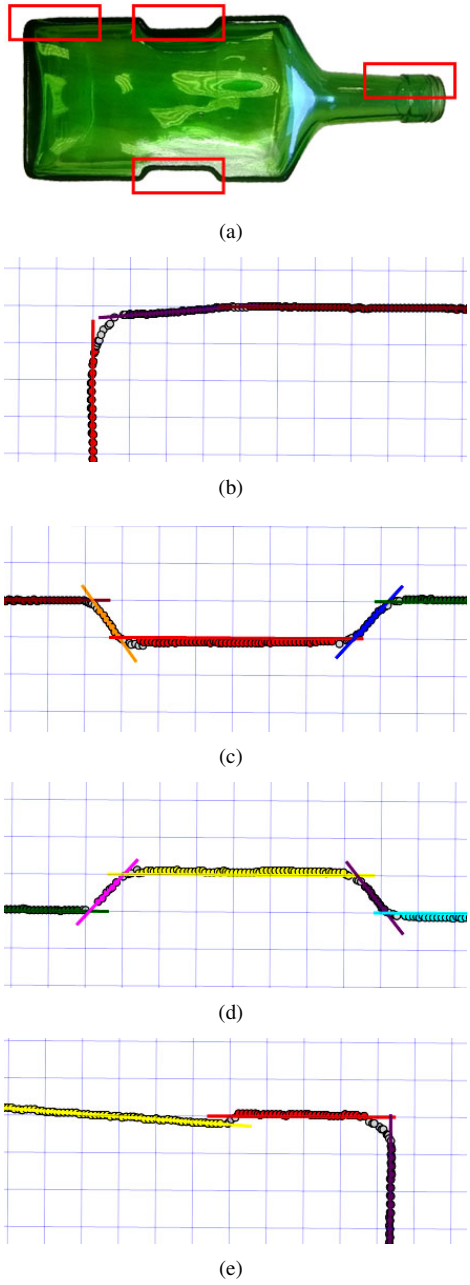


Figure 8: Close views of Figure 5.

7 Conclusion

It is a crucial issue to perfect engineering objects being reconstructed from measured data. Having only rough approximations for perpendicularity, parallelism, concentricity, etc, would not be acceptable for the majority of downstream CAD applications. In this project we have investigated *perfecting techniques* to automatically set up and enforce local and global geometric constraints. We have tested our methods for planar point data sets, represent-

ing straight and circular curve segments. These algorithms can be generalized to 3D objects in a reasonably straightforward way; this is subject of ongoing research. In the future, we plan to detect other types of global symmetries, such as, rotational and translational symmetries, then later apply these techniques to couple conventional and free-form curves and surfaces, as well.

Acknowledgements

I would like to thank my supervisor Dr. Tamás Várady for many constructive discussions, and guiding me to write this article. I would also like to thank the other members of our research team – Péter Salvi, György Karikó and Pál Benkő – for exchanging important technical ideas. This research is supported by the Hungarian Scientific Research Fund (OTKA No.101845).

References

[1] P. Benkő, G. Kós, T. Várady, L. Andor, and R. R. Martin. Constrained fitting in reverse engineering. *Computer Aided Geometric Design*, 19(3):173–205, 2002.

[2] I. Coope. Circle fitting by linear and nonlinear least squares. *Journal of Optimization Theory and Applications*, 76(2):381–388, 1993.

[3] M. Li, F. C. Langbein, and R. R. Martin. Detecting approximate incomplete symmetries in discrete point sets. In *Proceedings of the 2007 ACM symposium on Solid and physical modeling*, pp. 335–340. ACM, 2007.

[4] Y. Li, x. Wu, y. Chrysathou, A. Sharf, D. Cohen-Or, N. J. Mitra. Globfit: Consistently fitting primitives by discovering global relations. *In ACM Transactions on Graphics (TOG)* (Vol. 30, No. 4, p. 52). ACM, 2011.

[5] G. Lukács, R. R. Martin, and D. Marshall. Faithful least-squares fitting of spheres, cylinders, cones and tori for reliable segmentation. In *Computer Vision-ECCV'98*, pp. 671–686. Springer, 1998.

[6] P. Marks. Capturing a Competitive Edge Through Digital Shape Sampling & Processing (DSSP). *SME Blue Book Series*, 2005.

[7] N. J. Mitra, L. J. Guibas, and M. Pauly. Partial and approximate symmetry detection for 3D geometry. *ACM Transactions on Graphics (TOG)*, 25(3):560–568, 2006.

[8] J. Porrill. Optimal combination and constraints for geometrical sensor data. *The International Journal of Robotics Research*, 7(6):66–77, 1988.

[9] H. Pottmann, S. Leopoldseder, and M. Hofer. Approximation with active B-spline curves and surfaces. In *Computer Graphics and Applications, 2002. Proceedings. 10th Pacific Conference on*, pp. 8–25. IEEE, 2002.

[10] R. Schnabel, R. Wahl, R. Klein. Efficient RANSAC for Point-Cloud Shape Detection. In *Computer Graphics Forum* (Vol. 26, No. 2, pp. 214-226). Blackwell Publishing Ltd. 2007.

[11] V. Schomaker, J. Waser, R. T. Marsh, and G. Bergman. To fit a plane or a line to a set of points by least squares. *Acta crystallographica*, 12(8):600–604, 1959.

[12] T. Várady and R. R. Martin. Reverse engineering. *G. Farin, J. Hoschek, M. S. Kim, Handbook of Computer Aided Geometric Design, Chapter 26*, Elsevier, 2002.

[13] T. Várady, P. Salvi, *3D Geometric Modelling and Digital Shape Reconstruction, Lecture Notes*, Budapest University of technology and Economics, BME IIT, 2013.

[14] N. Werghi, R. Fisher, C. Robertson, and A. Ashbrook. Modelling objects having quadric surfaces incorporating geometric constraints. In *Computer Vision-ECCV'98*, pp. 185–201. Springer, 1998.

Attention and Metrics

Modified Methods of Generating Saliency Maps Based on Superpixels

Veronika Olešová*
Supervised by: Ing. Vanda Benešová, PhD.

Institute of Applied Informatics
Faculty of Informatics and Information Technologies STU
Bratislava

Abstract

Various types of approaches that can model a human visual attention have been already proposed. However, a model that could perfectly simulate the human perception and methods of computer prediction of human visual attention belongs to one of the high focused research areas.

Our work is aimed at methods of generating a saliency map which will detect the areas in the picture that could most likely attract a human attention. The basis of our work is method of saliency detection using superpixels published by authors Z. Liu, O. Meur and S. Luo. Several modifications of this method with the aim to improve the results have been proposed, tested and evaluated in our experiments. In this paper, we present our proposed modification based on border prior and statistical evaluation of the saliency central position in the used dataset. This center position will be expressed using a fitted Gaussian function. Results of all experiments are evaluated and presented in the following paper.

Keywords: saliency map, superpixels, border prior

1 Introduction

In our daily lives we are surrounded by incredible amount of information, which we are not able to process all at once. We need to restrict our attention only on certain area or objects at a time so we can process this information one after another. Scientists have been examining what underlies our attention to help us avoid information overload. They came up with the idea to create a saliency map for a given image that represents information about human visual attention of this image.

The saliency map is a topographically arranged map to represent the saliency of the visual scene and it gives us information about where in the image the areas that attract our attention are. It can reflect several salient objects or areas which are sorted by their saliency.

Saliency map is often used as a prior for a classification system to detect objects. These maps are useful for many applications such as image compression, predicting eye movements, autofocus and visualization.

The main problem of existing models generating saliency maps is that they usually work with specific cases and are not able to cover all of them.

2 State Of The Art

There are a lot of differently oriented models to creating a saliency map that have achieved good performance in predicting human eye fixations. The most common models that are often used for comparison are A Model of Saliency-based Visual Attention for Rapid Scene Analysis [5], Graph-Based Visual Saliency [4] and SUN: A Bayesian Framework for Saliency Using Natural Statistics [10]. We will describe the main ideas of these models in this section. In more detail we will analyze another model, called Superpixel-based saliency detection [8], which is the basis of our work.

2.1 Model of Saliency-based Visual Attention for Rapid Scene Analysis

Itti proposed a model [5] which is inspired by the architecture proposed by Koch and Ullman, who came up with the idea that the different visual features should be combined into one single topographically oriented map. Most of the later works use Ittis model for comparison since it is the earliest model of a saliency map.

Visual preprocessing of this model consists of creating five Gaussian pyramids that are generated from intensity image and four color channels - red, green, blue and yellow. Image is then decomposed into a set of topographic feature maps. Each feature is computed by a set of linear center-surround operations. These maps are normalized and combined into three conspicuity maps. The final saliency map is the result of the normalization followed by a summation of the three conspicuity maps.

This architecture is not designed to detect conjunctions of features; it can only recognize a target which is different from surrounding by its intensity, color, size and orientation, and will fail once the salient object has another

*v.nika.olesova@gmail.com

feature. The salient object has to be represented in at least one feature map in order to pop out.

2.2 SUN: A Bayesian Framework for Saliency Using Natural Statistics

The title of the second mentioned approach [10] is SUN because it depends on the statistics of natural images. The saliency map of this framework can be generated either by bottom-up, top-down or a combination of those approaches. By choosing bottom-up approach, saliency is represented by self-information and by choosing top-down, it is defined as log-likehood. In this model, the features are calculated in two ways. The first approach calculates the features as responses of linear filter known as DoG and the second as the responses to filters learned from natural images using independent component analysis ICA.

2.3 Graph-Based Visual Saliency

Graph-Based Visual Saliency [4] consists of three main steps. First, feature maps need to be extracted at multiple spatial scales. To do that, a scale-space pyramid is obtained from image features: intensity, color and orientation, which is similar to model of Itti. The second step is to form an activation map using these feature maps. In the final step the activation map is normalized to emphasize the most important information and then combined into a single saliency map.

This model assigns greater saliency to locations situated in the middle of the image. The reason is that most of nodes are closer to a few center nodes than to any point located near the image boundary. The described process is computationally quite expensive and the resulting saliency map has ill-defined object boundaries, which can restrict the usefulness in certain applications.

2.4 Superpixel-based saliency detection

This model [8] consists of three major steps. At the beginning it is important to simplify the input image by using superpixel segmentation and color quantization. Then, similarity between each superpixel has to be found. Finally, the global contrast and spatial sparsity is computed for each superpixel.

A superpixel should contain pixels that are similar in color and texture, and therefore are likely to belong to the same object. This assumption leads to the advantage of superpixel primitives over pixel primitives. Another advantage of this representation is that computational elements are greatly reduced and the segmentation result will be better since superpixels preserve the objects shape information and are more robust to noise.

2.4.1 Image simplification

The image is converted to the CIE L*a*b*, perceptual uniform color space, which is designed to approximate human vision. The first simplification consists of creating superpixels using SLIC algorithm [2]. This divides a picture into approximately 200 smaller regions, which is sufficient to preserve different boundaries in the used dataset well. The result of superpixel segmentation using SLIC algorithm can be seen in Figure 1. Then, the number of distinct colors has to be reduced by applying the color quantization. The image histogram is created by quantizing each color into *qxqxq* bins. For each bin, mean color and number of pixels belonging to this bin is computed. Bins that cover more than certain number of pixels are preserved and the rest are merged into ones that have the smallest difference between their quantized colors.

Figure 1: Superpixel segmentation

2.4.2 Superpixel similarity

Each superpixel is assigned to a color histogram which is calculated based on the one created in the previous step. The histogram is normalized so that the summation of values in each histogram is equal to 1. Two types of similarities for each pair of superpixels are computed.

The color similarity of two superpixels is computed as the sum of intersection of their histograms:

$$Sim_c(i,j) = \sum_{k=1}^{m} min\left\{H_i(k), H_j(k)\right\} \tag{1}$$

The spatial similarity is defined as:

$$Sim_d(i,j) = 1 - \frac{\left\|\mu_i - \mu_j\right\|}{d} \tag{2}$$

where d is the diagonal length of the image and μ is the center of the superpixel.

By combining those similarities, the resulting similarity is obtained:

$$Sim(i,j) = Sim_c(i,j) * Sim_d(i,j) \tag{3}$$

2.4.3 Superpixel saliency

Authors [8] suggested that color contrast can be easily seen between the salient object and the background. They also noticed that spatial distribution of salient object superpixels is sparser than background superpixels. Because of this, global contrast of each superpixel and their spatial sparsity are evaluated for measuring the final saliency.

Global contrast of each superpixel is defined as:

$$GC(i) = \sum_{j=1}^{n} W(i,j) \cdot \left\| mc_i - mc_j \right\| \tag{4}$$

where *mc* is the mean color of superpixel and the weight is defined as:

$$W(i,j) = \mid SP_j \mid \cdot Sim_d(i,j) \tag{5}$$

where $\mid SP_j \mid$ stands for the number of pixels in the superpixel. We have to normalize this global contrast so that the values map to the range from 0 to 1:

$$NGC(i) = \frac{GC(i) - GC_{min}}{GC_{max} - GC_{min}} \tag{6}$$

where GC_{max} is the maximum value of global contrast among all the superpixels.

The spatial sparsity of a superpixel is computed as:

$$SS(i) = \frac{\sum_{j=1}^{n} Sim(i,j) \cdot D(j)}{\sum_{j=1}^{n} Sim(i,j)} \tag{7}$$

where $D(j)$ is a distance between the center of image and the superpixel j. This is also normalized, but this time inversely:

$$NGC(i) = \frac{GC(i) - GC_{min}}{GC_{max} - GC_{min}} \tag{8}$$

We have refined the normalized global contrast and spatial sparsity so that superpixels with higher similarity have more similar values:

$$RGC(i) = \frac{\sum_{j=1}^{n} Sim(i,j) \cdot NGC(j)}{\sum_{j=1}^{n} Sim(i,j)} \tag{9}$$

$$RSS(i) = \frac{\sum_{j=1}^{n} Sim(i,j) \cdot NSS(j)}{\sum_{j=1}^{n} Sim(i,j)} \tag{10}$$

The final saliency value for each superpixel is defined as the multiplication between refined global contrast and spatial spread:

$$Sal(i) = RGC(i) * RSS(i); \tag{11}$$

3 Our Contribution

In this section we present our experiments that include border prior, its update and central position modification.

3.1 Border Prior

We have extended the original model by adding the border prior, which achieves better results. This prior comes from the basic rule of photographic composition, that is, most photographers will not crop salient objects along the view frame. In other words, the image boundary is mostly background [9]. However, this only applies to photographs that are intentionally taken by humans and it is not general.

Huaizu Jiang and others [6] made the following survey: "we made a simple survey on the MSRA-B data set with 5000 images and found that 98% of pixels in the border area belong to the background."

In our algorithm we label the superpixels that touch any of the image borders as background and find other superpixels that are very similar to them. Each of these superpixels is considered background and its saliency is automatically zero. In the Figure 2 we can see the difference between the saliency map which uses this prior and the one that does not.

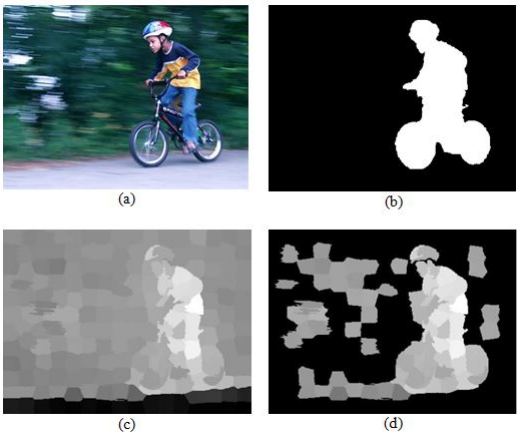


Figure 2: (a) Original image, (b) ground truth, (c) saliency map without border prior, (d) saliency map with border prior.

However, if there is a salient object that only slightly touches the boundary, the whole object could be missed. In order to prevent such situation, we compute global contrast for the group of superpixels that touch the boundary and remove the first 10% whitest of them. These superpixels could be a part of the object and it would be wrong to mark them as background. The chosen percentage is only an estimation based on observation of the used dataset of images. An example is shown in Figure 3 where we can see that in the image c) a man is missed because he touches the boundary and in the image d) we see that the most contrast superpixels help us identify this man.

3.2 Central Position Modification

The original distance shown in Equation 7, which is used for computing the spatial sparsity, did not seem accurate to us. The result of the function $D(j)$ is simply a distance

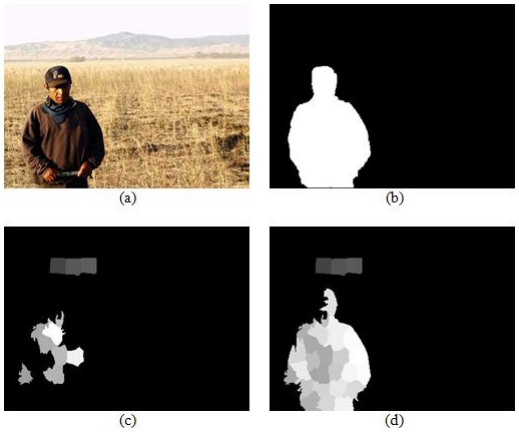


Figure 3: (a) Original image, (b) ground truth, (c) saliency map border prior, (d) saliency map with updated border prior.

between a superpixel j and center of image. It does not take into account the most probable distance to which the salient object could occur. We decided to statistically evaluate the central position in the dataset and create a new function that could be used instead of the original distance. This modification is also not general and applies only to the used dataset.

Firstly, our intention was to get a histogram for each ground truth image in the dataset, which would indicate how far is the salient object from the center. Ground truth is human-segmented image dataset used to compare image segmentation algorithms. Basically, it is a binary image whose white pixels belong to the salient object and black pixels to background. To create a histogram, we calculated the number of white pixels that fall within each distance from the center position of the image. Then we summed all the histograms of each image and divided each value of the resulting histogram by the length of the corresponding circle. This histogram was then fitted to Gaussian function using matlab:

$$[fy, god] = fit(x, y,' gauss1'); \tag{12}$$

where x is a vector of distances from the center image and y is a vector of number of pixels.

The plot of the resulting function is in the Figure 4 where we see that most of the pixels belonging to the object are situated near the center of the image and the output is the Gaussian function in the following form:

$$726.9 * exp(-(\frac{x - 6.692}{97.7})^2) \tag{13}$$

The distance $D(j)$ has been replaced by this exponential function.

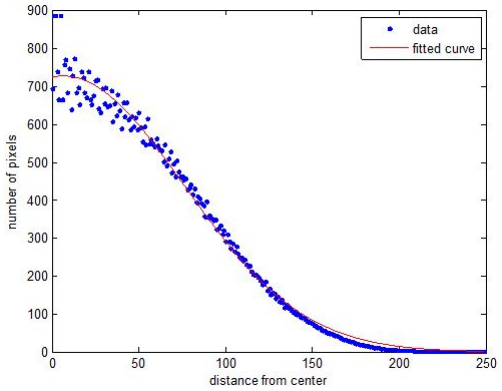


Figure 4: Fitted gaussian function.

4 Tests and Results

We came up with two types of evaluation. In each of them we use images from the MSRA[1] dataset, which is the largest object dataset containing 20 000 images in set A and 5 000 images in set B. Achanta [1] has created the dataset[2] containing 1 000 manually segmented ground truths corresponding to 1 000 images from the set B.

4.1 Precision and Recall

The first type of evaluation is used to test a precision and recall of a border prior, its update and a center modification. Precision and recall are statistical measures that are very often used to measure how well the saliency model is able to predict human eye fixations. Precision is a measure of accuracy and recall is a measure of completeness.

At first we have to generate a saliency map for each of 1000 images from MSRA dataset. To get a segmented image we simply threshold the map by assigning the pixels above the given threshold as salient (white background) and below the threshold as non-salient (black background). Then we compare the resulting image to its ground truth. From this comparison we are able to get statistics like precision and recall rate by using the following pseudo-code:

```
if (value_of_saliency_map > threshold)
   {
        segmented_foregound_pixels++;
        if (value_of_ground_truth != 0)
            hit++;
   }

if (value_of_ground_truth != 0)
    ground_truth_foreground_pixels++;

precision = hit / segmented_foregound_pixels;
recall = hit / ground_truth_foreground_pixels;
```

By sliding the threshold from minimum to maximum

[1]Downloaded from http://research.microsoft.com/en-us/um/people/jiansun/SalientObject/salient_object.htm

[2]Downloaded from http://ivrgwww.epfl.ch/supplementary_material/RK_CVPR09/

value, we achieved the precision-recall curves that we use for the comparison between various methods.

The graph of comparison between the algorithm without and with the border prior implemented (SB and BP) is in Figure 5. We can see that our algorithm updated with the border prior achieves better results in precision. There is no saliency map that would have the precision smaller than 0.55. In addition, this graph shows the difference between another 3 models including Graph-Based Visual Saliency (GB) [4], A Model of Saliency-based Visual Attention for Rapid Scene Analysis (IT) [5] and Frequency-tuned Salient Region Detection (IG) [1]. To compare these methods subjectively, we created a table of few images that can be found in the Figure 6. We can see that the background is most suppressed using the border prior. In this comparison we used datasets containing 1000 saliency maps for each model created by Achanta et al.

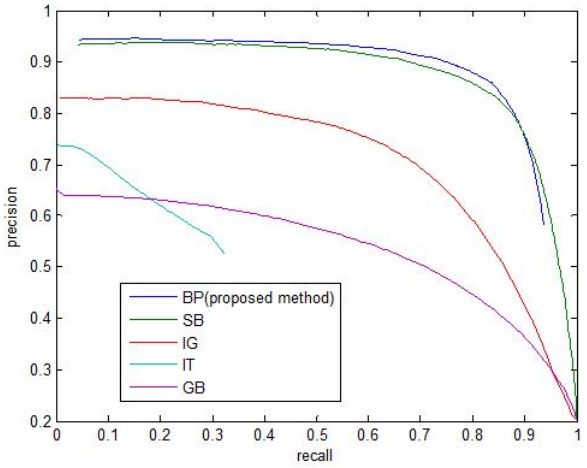


Figure 5: Comparison between different saliency models.

Figure 6: (a) Original image, (b) IT, (c) GB, (d) IG, (e) original, (f) border prior, (g) ground truth.

In the second graph represented by Figure 7, we can see a precision-recall curve between border prior (BP) and its updated version (UBP). Unfortunatly, precision of this method has regressed but the recall has improved. Images with the object touching the boundary were succesfully identified, however, this dataset contains a lot of pictures without such objects. In those images, by removing 10% of superpixels from background we removed superpixels that were actually background.

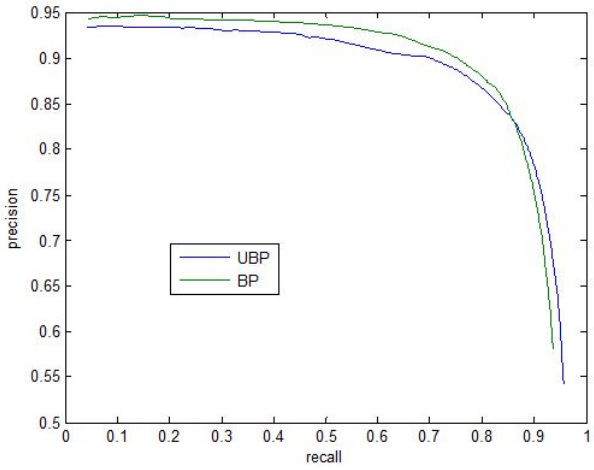


Figure 7: Comparison between border prior (BP) and updated border prior (UBP).

We have also evaluated the modification of center position (BPCM) which can be observed in Figure 8. The recall rate of this modification is the same as the unmodified border prior but the precision has decreased. We assume that its because of the images that do not fit into our gaussian function.

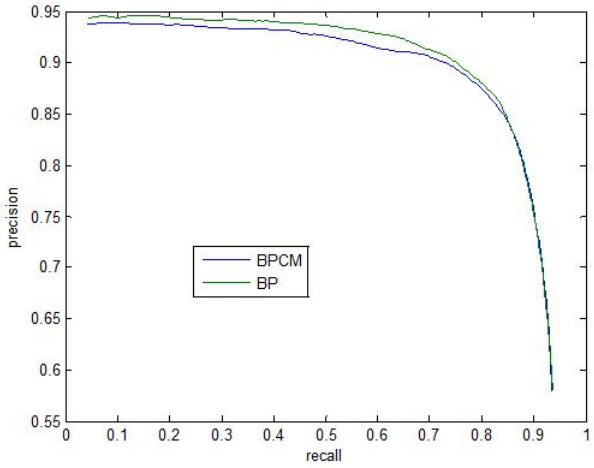


Figure 8: Comparison between border prior (BP) and border prior with center position modification (BPCM).

4.2 Histograms

The second evaluation is implemented in matlab and is aimed at any of the modification but we used it to test the updated border prior. The key is to create a histogram by which we would be able to see how many pixels and what shades of gray from our saliency map belong to the object

and how many to background. This is done by comparing our saliency map to the ground truth. The number of pixels that belong to salient object and to background are computed individually. We divided grayscale into 10 intervals and assigned a number of corresponding pixels from our saliency map to each of them. Therefore each bar of histogram is an interval of size 25 and holds a number of pixels.

An example of such histogram is in the Figure 9, which evaluates the images (c) and (d) in the Figure 3. A symbol TP in this histogram stands for the true positive (number of pixels belonging to the object) and FP is false positive (number of pixels belonging to background). We can see that TP - original (image (c) in the mentioned figure) bar with the pixel value of 1 is bigger than bar TP - modification (image (d)) next to it. This means, that the image with only border prior implemented (TP - original) has more pixels in the range between values 0-25 belonging to object. The other method d(TP - modification) has this number lower, which is good, because we do not want black pixels in the object.

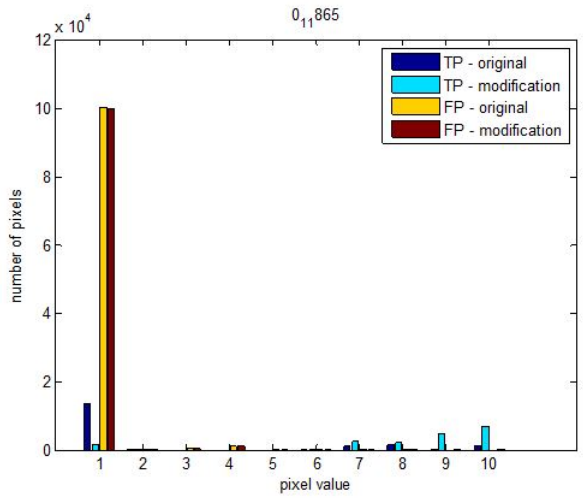


Figure 9: Comparison between border prior (original) and its updated version (modification) by histogram.

5 Conclusion and Future Work

We have presented a few modifications to the existing method [8] to creating a saliency map. These modifications are customized to the used dataset and therefore are not general. Comparing to other models using the same dataset we were able to see that our modification of border prior is better at precision but slightly worse in recall. The update to this method slightly downgrades the precision but improves recall and modification of center position does not change the recall of the border prior but decreases precision.

However, results provided by this method are still not perfect and other modifications are required. We assume that using only color contrast, spatial distribution and border prior is not enough and it would be vital to use higher features such as face detection. Our next goal is to implement a center surround method adjusted to superpixels. Authors suppose that the salient object is enclosed by a rectangle R and they construct a surrounding contour R_s with the same area of R. Then the distance between R and R_s can be measured using various features such as intensity, color, and texture. By this technique it is possible to measure how distinct the salient object in the rectangle is with respect to its surroundings. In our case we would use groups of superpixels instead of rectangles and measure a distance between these groups and their surrounding superpixels by color.

Images in the MSRA dataset contain only a single salient object and most of them are large and near the image center. For the future work it would be appropriate to use more challenging images in a combination with a dataset containing human eye fixations.

References

[1] R. Achanta, S. Hemami, F. Estrada, and S. Süsstrunk. Frequency-tuned salient region detection. *Proc. IEEE CVPR*, pages 1597 – 1604, June 2009.

[2] R. Achanta, A. Shaji., K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk. Slic superpixels. *EPFL Technical Report*, (149300), June 2010.

[3] R. Gonzalez and R. Woods. *Digital Image Processing.* Number 2. 2001.

[4] J. Harel, C. Koch, and P. Perona. Graph-based visual saliency. *Advances in Neural Information Processing Systems 19*, pages 545–552, 2007.

[5] L. Itti, C. Koch, and E. Niebur. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(11), 1998.

[6] H. Jiang, J. Wang., Z. Yuan, Y. Wu, N. Zheng, and S. Li. Salient object detection: A discriminative regional feature integration approach. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2083–2090, 2013.

[7] T. Liu and Z. Yuan. Learning to detect a salient object. *IEEE transactions on pattern analysis and machine intelligence*, 33(2):353–367, February 2011.

[8] Z. Liu, O. Meur, and S. Luo. Superpixel-based saliency detection. *International Workshop on Image and Audio Analysis for Multimedia Interactive services*, pages 1–4, July 2013.

[9] Y. Wei, F. Wen., W. Zhu, and J. Sun. Geodesic saliency using background priors. *Proceedings of the 12th European Conference on Computer Vision*, pages 29–42, 2012.

[10] L. Zhang, M. H. Tong, T. K. Marks, H. Shan, and G. W. Cottrell. Sun: A bayesian framework for saliency using natural statistics. *Journal of Vision*, 8(7):1–20, 2008.

Gaze-dependent Ray Tracing

Adam Siekawa*
Supervised by: Radoslaw Mantiuk†

Institute of Computer Graphics
West Pomeranian University of Technology
Szczecin / Poland

Abstract

In this paper we introduce a method for acceleration of the real time ray tracing by using characteristic traits of visual perception. Ray tracing is a demanding rendering technique which is much slower than currently dominating scanline methods. Performance hit especially arise when we use huge amount of samples for anti-aliasing or other sample-based effects. We show how to decreases number of rays by increasing the perceptual size of selected pixels by using combination of eye tracking and the human gaze-dependent contrast sensitivity. Our study shows that number of processed pixels can be reduced three times without perceptually noticeable quality loss. As a result, we can greatly increase performance of ray tracing.

1 Introduction

The gaze-dependent vision is a characteristic way in which the human visual system builds an image of the world. We perform frequent and rapid eye movements, called saccadic movement, and follow moving objects in the smooth pursuit movement [2]. These rapidly changing snapshots are combined by the Human Visual System (HVS) in a stable image of the entire scene. Interestingly, a gaze-dependent model of image synthesis is not used in contemporary computer imaging systems, even despite the fact that a significant reduction of computation complexity is possible during image rendering in the parafoveal and peripheral regions of vision.

Ray tracing is a popular image synthesis technique which can benefit from the gaze-dependent characteristic of vision. Generally, even using the basic Whitted model [11], the ray tracing can produce images of the higher quality that the scanline techniques. However, this is achieved at the expense of larger computation complexity. The main bottleneck of this technique - finding intersections - can be reduced by decreasing the number of primary rays. In this work we propose a gaze-dependent ray tracing in which the number of rays per unit angle fits the sensitivity of HVS. We use the gaze-dependent contrast sensitivity function (CSF) to reduce sampling in peripheral vision. Temporal location of the gaze point is captured by the eye tracker and used by the interactive ray tracing system to render images with the highest quality only in the gaze-point surrounding.

In Sect. 2 we outline the directionality of the human vision, introduce the gaze-dependent CSF and discuss existing gaze-dependent techniques of image synthesis. Sect. 3 presents our gaze-dependent ray tracing system based on the weighted sampling. In Sect. 4 we show how the gaze-dependent sampling was implemented in our ray tracer. Sect. 5 discusses the achieved performance boost with respect to image quality deterioration. The paper ends with conclusions and future work in Sect. 6.

2 Background

In this Section we present a basis of the human eye physiology and describe technologies used in the gaze-dependent rendering frameworks.

2.1 Gaze-dependent contrast sensitivity function

Human vision has a strong feature of the directionality of view. We can see the details only in a small viewing angle subtended 2-3 degrees of the field of view. In this range, a human sees with a resolution of up to 60 cycles per angular degree, but for a 20-degree viewing angle, this sensitivity is reduced ten times [6].

The fundamental relationship describing the behaviour of the human visual system is the contrast sensitivity function (CSF) [1]. It shows the dependence between the threshold contrast visibility and the frequency of the stimulus. The CSF can be used to e.g. better compress the image by removing the high frequency details that would not be seen by humans. An extension of the CSF, called the gaze-dependent CSF, is measured for stimuli observed in various viewing angles [3, 12]. It models the impact of deviations from the axis of vision (called eccentricity (E)) to the most recognisable stimulus frequency (see Fig. 1).

In this work we use the gaze-dependent CSF proposed by Peli et al. [3]:

$$C_t(E,f) = C_t(0,f) * exp(kfE), \qquad (1)$$

*adamsiekawa@gmail.com
†rmantiuk@wi.zut.edu.pl

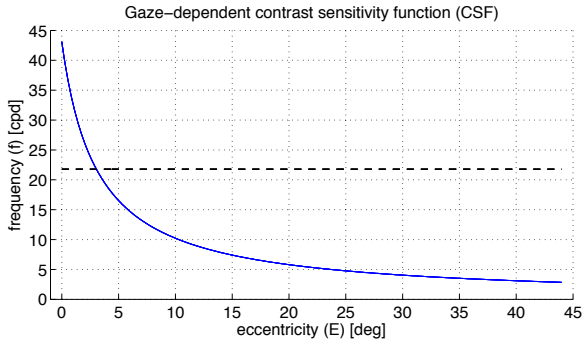


Figure 1: Gaze-dependent contrast sensitivity function. The dashed line shows the maximum frequency of our display.

where C_t denotes contrast sensitivity for spatial frequency f at an eccentricity E, k determines how fast sensitivity drops off with eccentricity (the k value is ranged from 0.030 to 0.057). Based on the above equation, the cut-off spatial frequency f_c can be modelled as:

$$f_c(E) = min(max\_display\_cpd, E_1 * E_2/(E_2 + E)), \quad (2)$$

where E_2 is retinal eccentricity at which the spatial frequency cut-off drops to half its foveal maximum (it ranges from E_1=43.1 to 21.55), and $E_2 = 3.118$ (see details in [13]). An example region-of-interest mask computed for our display based on the gaze-dependent CSF is presented in Fig. 2. Applying this mask, one can e.g. sample an image with varying frequency generating less sampling rays for the peripheral regions of vision.

Figure 2: Region-of-interest mask computed based on CSF for an image of 1920x1080 pixel resolution (gaze position at $(1000, 500)$), lighter area denotes higher frequency of HVS.

2.2 Gaze-dependent image synthesis

Information about temporary gaze direction was previously used to reduce the computational complexity of the image synthesis. An example of this approach is the technique called the level of detail (LOD), in which the simplified models of objects are located in the peripheral areas of vision [7, 9].

In the ray casting [9] and volumetric rendering [5] the gaze-dependent sampling is applied in the screen space.

A similar solution was used to accelerate the ambient occlusion algorithm [8]. This technique introduces a novel filtration method, in which the global lighting is calculated only for the area surrounding the gaze point. In peripheral areas of vision only fast computations based on the local lighting model are performed. The perceptual experiments showed that the participants did not notice the quality deterioration of the generated images.

The models of the gaze-dependent vision seems to gain an increasing importance in improving the efficiency of the image synthesis. The leading IT companies are interested in new gaze-dependent rendering techniques. For example, in the solution proposed by Gunter and others in [4], the scanline-based rendering engine generates three low-resolution images corresponding to the different fields of view. Then, the wide-angle images are magnified and combined with non-scaled image of the area surrounding the gaze point. Thus, the number of processed pixels can be reduced by 10-15 times.

3 Gaze-dependent rendering

Fig. 3 presents the gaze-dependent rendering scheme. Our system requires the eye tracker data which represents a momentary gaze direction of a human observer. We render the scene using ray tracing. The screen space is sampled (the primary rays are generated) according to the gaze-dependent contrast sensitivity function. Less rays is generated in parafoveal and peripheral regions. The output image is reconstructed from the non-uniformly distributed samples and displayed in real time on the screen.

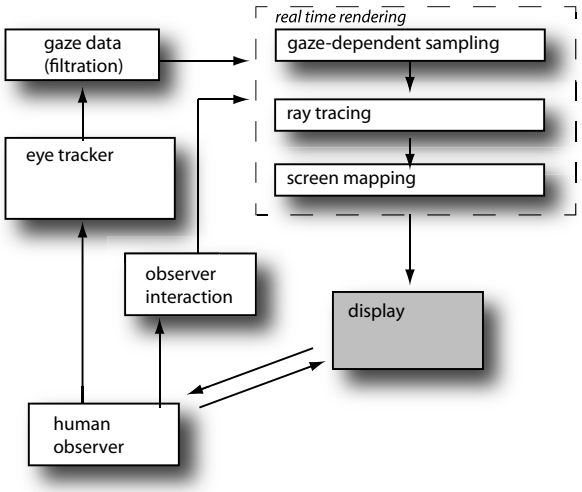


Figure 3: Gaze-dependent rendering system.

The whole system must be scaled in the real-world di-

mensions. We transform the gaze data to screen space using physical dimensions of a display, its resolution, and viewing distance between observer and the display screen.

3.1 Gaze-dependent sampling

The gaze-dependent CSF defines a solid angle in which a human cannot see details. This angle defines a limit of the HVS resolution and can be scaled in the perceptual JND units. We call this angle a perceptual unit angle. The further from gaze point a sample is, the larger the angle becomes. In this angle the human eyes integrate the image, i.e. it computes the average luminance. To sample an image during rendering, we use the constant number of rays per perceptual unit angle. For peripheral vision, the perceptual unit angle covers more pixels and the number of rays per image area decreases (see Fig. 4).

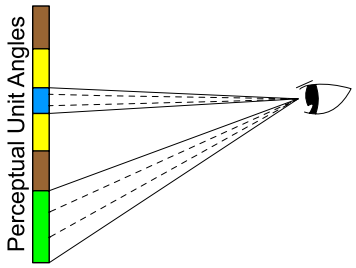


Figure 4: Less sampling rays falls on the area in the peripheral vision. The perceptual unit angles are marked in colours.

The perceptual unit angle (α) covers an area derived from CSF:

$$\alpha = 1/(2 * E_1 * \frac{E_2}{E + E_2})[deg], \quad (3)$$

where E denotes a viewing angle subtended from the gaze direction to the direction towards a considered pixel, E_1=43.1, E_2=3.118 [3]. This angle can be computed using equation:

$$E = atan(\frac{p_{distance} * p_{size}}{d})[deg], \quad (4)$$

where $p_{distance}$ is a distance between pixel and the gaze point in [pixels], p_{size} is a physical pixel size in [cm], and d denotes a distance from the screen to observer's eyes expressed in [cm].

The number of pixels covered by a perceptual unit angle α can be derived from:

$$\rho = \| \frac{\alpha}{\beta} \|, \quad (5)$$

where β is viewing angle in [deg] corresponding to one pixel.

In the gaze dependent renderer one can reduce the number of rays shoot per pixel based on the information whether a considered pixels belongs to the larger or smaller perceptual unit angle.

In our system we group together pixels belonging to one perceptual unit angle and form cells. Then, the image is sampled based on distribution of the cells. We shoot the constant number of anti-aliasing rays per cell (see details in Sect. 4.2) but, as the cells are larger in peripheral vision, the total number of sampling rays is reduced. Cells positioned further from gaze point will produce less anti-aliased results, however the artefacts will not be visible for the human observer. Cells are put together into an image after rendering. Visual representation of cell distribution is presented in Fig. 5.

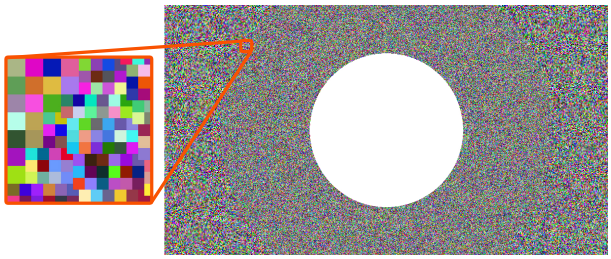


Figure 5: Cell map generated for an example location of the gaze point. Each cell larger than original pixel size is given a random color, non scaled cells are coloured in white. The enlargement shows unique structure of the map.

3.2 Rendering

The ray tracing was used for rendering because of simplicity of implementation of complex sampling schemas in the screen space. We implemented the Whitted ray tracing model which supports Phong lighting, shadows, reflection and refraction rays. See the implementation details in Sect. 4.1.

In this work we use a prototype renderer which does not work in real time. This solution does not meet the main assumption of the gaze-dependent rendering system, i.e. the gaze-driven rendering in which image content is changed with the gaze movement. However, our setup allows to perform the quality tests based on the offline results.

In future work we plan to adapt a real time ray tracer, such as OptiX [10] or Octane Renderer. Alternatively, we consider implementation of own ray tracer engine based on OpenCL or CUDA APIs. In the raw estimation, one ray should be rendered in $3e^{-8}$ [sec] to generate 60 frames-per-second in a typical viewing conditions. This requirement seems to be a challenge for a typical ray tracer and the gaze-dependent solution which significantly reduces the number of traced rays is highly beneficial (see details in Sect. 5).

4 Implementation

We implemented our own ray tracer extended with the gaze-dependent sampling technique. All images presented in this paper were rendered with this application.

4.1 Ray tracer

Ray tracer that was used in our project as a proof-of-concept is an off-line renderer implemented in C Sharp. It is build in a content based fashion, where one can create material by adding extra effects to the base type. There are two lighting models implemented: Phong and Ashikhmin-Shirley models. The ray tracer supports reflections, refractions, textures, and both hard and soft shadows.

One can load 3D scene stored in most of the popular formats, e.g. Wavefront OBJ format, COLLADA, or Autodesk 3DS file. The ray tracer can also render non-triangulated spheres. A scene is created in a code, i.e. one can position loaded models, created spheres, lights and camera then append them to the rendering list. The octree acceleration structure is applied to improve performance. Moreover, C# style *parallel foreach* is used for sampling each cell individually and utilise all available CPU cores. Results are saved as a linearly tone-mapped bitmap image. It is also possible to output image sequence which can later be put into a video.

Our ray tracer implements stochastic, regular (samples are distributed in a grid fashion) and adaptive anti-aliasing techniques. However, we found the most useful the stochastic sampling based on the random samples distribution. We use this type of anti-aliasing in all tests. Sample rays are distributed to fit the whole cell region (see details in 4.2). The first ray is always shoot in the centre of the cell (pixel or group of pixels). For the following samples we generate random single precision value which is used for offsetting ray direction.

4.2 Cell map generator

A cell map generator is an implementation of the gaze-dependent sampling in which cells are the discrete representation of the perceptual unit angles (see Sect. 3.1). One cell can cover one or more pixels, as seen in Fig. 6. Our algorithm requires information about gaze point (obtained from eye tracking device) and viewing conditions (width, height and viewing distance from a display) to compute number of pixels ρ that is within perceptual unit angle size (see Sect. 3.1). Result of the cell map generator is a *cell vector* with one cell per one unit angle and a *cell mask* which stores relationship between cells and pixels.

Single cell is a structure described by the set of parameters:

- unique cell id stored also in the cell mask
- size ρ, when equal to 1 it indicates that cell is covering single physical pixel.

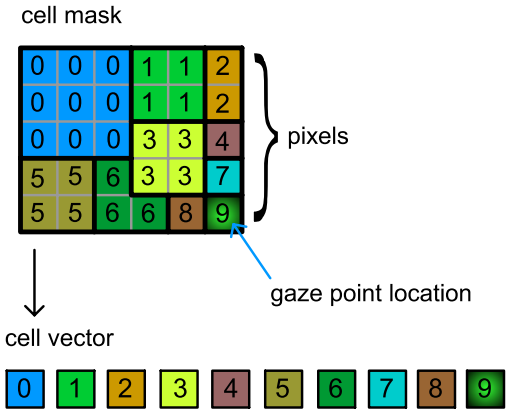


Figure 6: A cell vector for a 5x6-pixel image. Groups of pixels covered by the unit angle computed for a current distance from the gaze point (pixel with index of 9) are assigned to consecutive indices in the cell vector.

- pixel's centre position in the screen space
- camera information
- default luminance value (clear color)

The *cell id* is necessary for image reconstruction. *Size*, *position* and *camera data* is used during ray tracing procedure to generate the primary rays. The cell mask forms a matrix (with the size of destination image), which contains *cell ids* and helps to maintain the overlapping cells.

The cell vector is sent to ray tracing pipeline and is used during AA rays generation. We distribute the constant number of samples in a region covered by a given cell.

Final step is image reconstruction, for that we need to use cell mask mentioned earlier. As illustrated in Fig. 6, pixels have the same *cell id* as the cell that covers them. In order to retrieve our image, we need to iterate over that mask and extract the final color value from a cell with the same *cell id* and write it into a place holder for an image (e.g. DirectX or OpenGL texture).

5 Results

We rendered a set of images applying the gaze-dependent sampling calibrated for our hardware setup: 1080p resolution display measuring a 50 [cm] screen width and 35 [cm] height, observed from 60 [cm]. We used 32 samples for the stochastic anti-aliasing. This number seems to generate almost perfectly anti-aliased images of our test scene. The computational complexity remains the same as in classic ray tracing and cell map generation is not taken into account since it is used as a precomputed input. Example renderings are presented in Fig. 7. In the top image a typical ray tracing technique with the per-pixel stochastic anti-aliasing was used. The bottom image was generated

using the gaze-dependent technique with 32 anti-aliasing rays per cell (perceptual unit angle). The quality of image with reduced number of samples is noticeably worse, however this deterioration is not visible if observer is looking at the gaze point. This phenomena is even better visible on video we delivered in the supplementary materials. We prepared a HDTV clip (1080p, 25 FPS) with sampling rate reduced to 4 anti-aliasing rays per a cell.

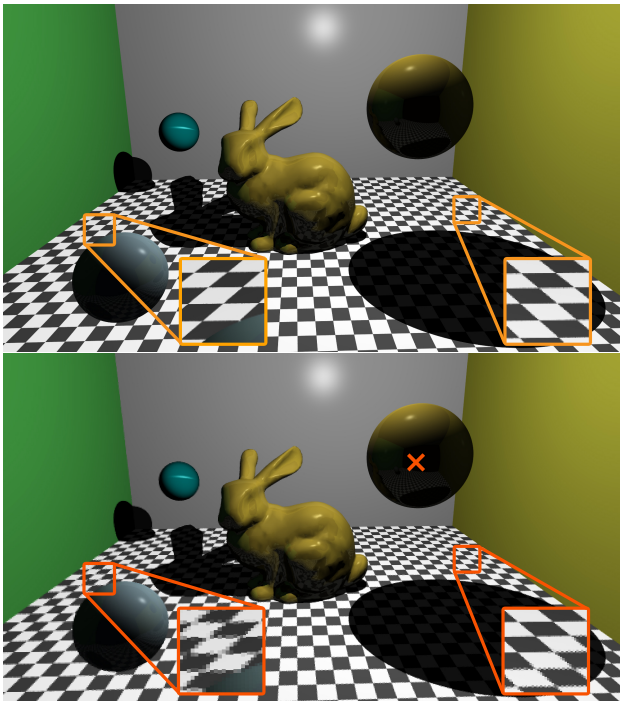


Figure 7: Comparison of the full frame (top) and the gaze-dependent rendering (bottom). Gaze point is marked as the red cross in the bottom image. The enlargements on the right depict borders between region with $\rho = 1$ and $\rho = 2$. Left enlargements show artefacts in region far away from the gaze position.

Cell overlapping

The cell mapping produces more apparent artefacts with increasing distance from the gaze point. Some of the cells may overlap each other, creating characteristic shapes similar to letter 'L', which are visible in Fig. 5. These artefacts are appearing when cell of size $\rho = n$ neighbours cell with size of $\rho = n - 1$, causing displacement of each consecutive cell. However, observer couldn't see this artefacts.

Performance

We measured the rendering performance on the laptop with Intel i3-2310M CPU, 2.1 GHz with 2 cores, 4 threads in total. It took 48 minutes and 16 seconds to render full frame anti-aliased image (see Fig. 7,top). During this time a 66.35 million anti-aliasing rays were traced. The cell map method needed only 14 minutes and 52 seconds with 20.65 million anti-aliasing rays shoot. The gaze-dependent technique was more than 3 times faster and almost 70% of sampling rays was required.

The acceleration will be even more significant for future displays of the retinal resolutions (60 cycles per visual angle). Our display should have a resolution of 5400x3900 pixels to reach the HVS resolution. In this case a typical full frame ray tracing would require 674 million sampling rays, but with cell map approach we would need only 27 million million rays, which is around 95% less.

6 Conclusions and Future Works

In this work we introduced gaze-dependent rendering as a sample reduction method for increasing ray tracing performance. Our algorithm is based on gaze-dependent CSF. It takes into account viewing conditions and physical dimensions of the display. We demonstrated how the cell mapping algorithm based on perceptual gaze-dependent sampling of the screen space can result in major performance boost. Although presented algorithm generates artefacts in the parafoveal region, they are unnoticeable for a viewer. In the paper we mainly focus on improving performance by accelerating anti-aliasing algorithms, but we expect that the same concept can be applied to other performance heavy algorithms based on sampling.

In future work we plan to deploy a real-time version of the system. In addition to the implementation of a fast ray tracer, our cell map generation process might proof to be difficult for parallel computing. One way of solving this issue is creating a precomputed cell map, which would use extra memory (four time more than map generated during runtime). We also want to address the problem of overlapping cells. Our algorithm might also proof useful in increasing performance of other rendering techniques i.e., path tracing or photon mapping.

References

[1] P. G. J. Barten. *Contrast sensitivity of the human eye and its effects on image quality*. SPIE Press, 1999.

[2] Andrew T. Duchowski. *Eye Tracking Methodology: Theory and Practice (2nd edition)*. Springer, London, 2007.

[3] Jian Yang Eli Peli and Robert B. Goldstein. *Image invariance with changes in size: the role of peripheral contrast thresholds*. JOSA A, Vol.8, Issue 11, 1991.

[4] Brian Guenter, Mark Finch, Steven Drucker, Desney Tan, and John Snyder. Foveated 3d graphics. *ACM Trans. Graph.*, 31(6):164:1–164:10, 2012.

[5] Marc Levoy and Ross Whitaker. Gaze-directed volume rendering. In *Proceedings of the 1990 symposium on Interactive 3D graphics*, I3D '90, pages 217–223, New York, NY, USA, 1990. ACM.

[6] L. C. Loschky, G. W. McConkie, J. Yang, and M. E. Miller. The limits of visual resolution in natural scene viewing. *Visual Cognition*, 12:1057–1092, 2005.

[7] David P. Luebke and Benjamin Hallen. Perceptually-driven simplification for interactive rendering. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 223–234, London, UK, UK, 2001. Springer-Verlag.

[8] R. Mantiuk and S. Janus. Gaze-dependent ambient occlusion. *Lecture Notes in Computer Science (Proc. of ISVC'12 Conference)*, 7431(I):523–532, 2012.

[9] Hunter A. Murphy, Andrew T. Duchowski, and Richard A. Tyrrell. Hybrid image/model-based gaze-contingent rendering. *ACM Trans. Appl. Percept.*, 5:22:1–22:21, February 2009.

[10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[11] Turner Whitted. An improved illumination model for shaded display. *Graphics and Image Processing*, 23(6):343–349, 1980.

[12] J. Yang, T. Coia, and M. Miller. Subjective evaluation of retinal-dependent image degradations. In *Proceedings of PICS 2001: Image Processing, Image Quality, Image Capture Systems*, Society for Imaging Science and Technology, pages 142–147, 2001.

[13] J. Yang, X. Qi, and W. Makous. *Zero frequency masking and a model of contrast sensitivity.* Vision Research, 1995.

An Experimental Study on Various Combinations of Shape Descriptors and Matching Methods Applied in the General Shape Analysis Problem

Katarzyna Gościewska *
Supervised by: Dariusz Frejlichowski†

West Pomeranian University of Technology, Szczecin,
Faculty of Computer Science and Information Technology,
Żołnierska 52, 71-210, Szczecin, Poland

Abstract

The aim of the General Shape Analysis (GSA) is to find one or a few most similar general templates for a processed object — exact identification is not performed, but the general shape features are obtained. The GSA approach may be applied for coarse separation of objects in the database prior to their classification or in the case when the data are incomplete or there is a high variability within them. In this paper, the GSA problem is investigated using various combinations of shape descriptors and methods for estimating the similarity or dissimilarity between shape representations. The experiments involved the use of five shape descriptors, namely the Two-Dimensional Fourier Descriptor, Generic Fourier Descriptor, UNL-Fourier descriptor, Zernike Moments and Point Distance Histogram, as well as four matching methods, that is the Euclidean distance, Mahalanobis distance, correlation coefficient and C1 metric. The effectiveness of the experiments was calculated as a coincidence between experimental results and results provided by people through inquiry forms. The experiments made it possible to determine the influence of various matching methods on the final effectiveness when a particular shape descriptor was applied. The best result was obtained for the combination of UNL-Fourier descriptor and C1 metric.

Keywords: General Shape Analysis, shape descriptors, similarity and dissimilarity measures

1 Introduction

The problem of General Shape Analysis (GSA) is considered similar to typical shape recognition or shape retrieval, however both of these approaches differ in their purpose. The GSA is aimed at finding one or a few most similar general templates for each investigated test object. Usually, a template is a simple geometrical figure, e.g. a triangle, rectangle or circle, whereas a test object is a more diversified shape for which the similarity to one or several templates is defined. This approach enables to determine the most general and predominant shape features. The idea of the GSA is to represent all shapes using a particular shape descriptor and calculate a similarity or dissimilarity measure between test objects and templates. Next, a set of most similar templates indicated by the algorithm is compared with the results provided by people through inquiry forms, in which they were asked to indicate five templates for each investigated object — from the most to the least similar one. The percentage convergence between the two gives the final effectiveness value of the experiment.

The General Shape Analysis was introduced in [1] and firstly applied for the Two-Dimensional Fourier Descriptor. In subsequent years, this approach has been examined with the use of other shape descriptors, among which were the UNL-Fourier descriptor [2], Generic Fourier Descriptor [3], Point Distance Histogram [2, 3], Zernike Moments [4], Moment Invariants [4] or Curvature Scale Space [5]. Based on what is known from available literature, the Euclidean distance has been used to establish shape dissimilarity. The first application of a different shape matching method was presented in [6], where the correlation coefficient was applied to match Fourier-based shape representations.

In this paper, various combinations of shape descriptors and matching methods are investigated. The shape descriptors include some of the methods mentioned above — the Two-Dimensional Fourier Descriptor, Generic Fourier Descriptor, UNL-Fourier descriptor, Zernike Moments and Point Distance Histogram, however in the paper several versions of each shape descriptor are used, i.e. feature vectors of various size. For shape matching, two dissimilarity measures were selected, namely the Euclidean distance and Mahalanobis distance, and two similarity measures — the correlation coefficient and C1 metric. This made it possible to determine the most appropriate method for solving the GSA task. Moreover, in this paper, a different approach for estimating experimental effectiveness

*kgosciewska@wi.zut.edu.pl
†dfrejlichowski@wi.zut.edu.pl

is applied. Usually, the three most similar templates indicated by the algorithm and people in the inquiry forms were compared with respect to the sequence of indications. According to the suggestions included in [4], the sequence of indications is not taken into account and only the first template indicated by the algorithm is considered. Under this condition a template is proper only if it matches one out of three indications from the human benchmark result.

Some may disagree with the abovementioned manner of estimating effectiveness, arguing that human shape perception is supported by the theory of Recognition-By-Components proposed by Biederman. This theory assumes that an object is an arrangement of a number of basic components, including block, sphere, arc, cylinder or wedge, and that these components can be used to describe a shape [10]. Furthermore, it cannot be overlooked that there are other approaches to cognitive pattern recognition, such as the Theory of Template or the Theory of Feature. According to the former, people store templates in the long-term memory and use them to recognize a pattern. Contrarily, the latter one states that instead of matching the entire pattern with a template, people try to match their features [15]. It also needs to be emphasized that the General Shape Analysis is not concerned with studying the way in which people process the shape and establish the similarity between some shapes, but it investigates the results provided by people and based on them it tries to find an appropriate substitute in the area of computer pattern recognition. In addition, we should also think of how people describe things in daily life. Relatively often we define a shape of an unknown object using commonly known features — for example, we say that something is round or square or has several features in the sense that we can distinguish several known characteristics in the entire shape. Moreover, in some aspects of life, establishing an objects similarity to simple geometric figures is common and considered useful, for instance, in case of human body shape or a shape of a face, where the awareness of the shape simplifies choosing an outfit or hairstyle. Despite the triviality of this example, it is undoubtedly true that people tend to compare shapes to their simpler equivalents.

Taking into account what has been stated in the above paragraph, as well as focusing on the algorithms, the GSA approach may be applied for coarse separation of objects in the database prior to their exact classification or in the case when the data are incomplete or there is a high variability among the data. The GSA has been successfully applied in the identification of stamp types, which is useful in searching for presumably falsified digital documents [14]. The approach may also be applied in searching large multimedia databases, where voice commands are used for shape retrieval [2]. In this paper, focus is not placed on a specific application, but rather on the evaluation of the methods and algorithms.

The rest of the paper is organised as follows. The second section describes methods for estimating similarity and dissimilarity between shape representations, i.e. methods for matching feature vectors. The third section briefly presents algorithms selected for shape representation. The fourth section provides the conditions of the experiments and experimental results concerning the application of various combinations of shape descriptor and matching method as part of the GSA task. The last section summarizes and concludes the paper.

2 Methods for estimating similarity and dissimilarity between shape representations

In the GSA, test objects are compared with the templates in order to estimate the similarity (or dissimilarity) between shapes. Shape similarity enables to establish the level of similarity (or dissimilarity) between two shapes. Shape similarity criteria have to be adapted to the specific problem, i.e. the shapes have to be represented using features relevant to the problem under consideration [8]. The similarity of shapes is determined through matching a shape and calculated measure. In this paper, four measures are investigated — two similarity and two dissimilarity measures. The similarity measure is based on the maximization of correlation between shapes, while the dissimilarity measure — on the minimization of the distance between shapes.

Let us take as an example two vectors $V_A(a_1, a_2, \ldots, A_N)$ and $V_B(b_1, b_2, \ldots, B_N)$, which represent object A and object B in a N-dimensional feature space. The Euclidean distance d_E between these two vectors is defined by means of the following formula [11]:

$$d_E(V_A, V_B) = \sqrt{\sum_{i=1}^{N} (a_i - b_i)^2}. \tag{1}$$

The Mahalanobis distance d_M between vectors V_A and V_B can be derived as follows [7]:

$$d_M(V_A, V_B) = \sqrt{(V_A - V_B)^T E^{-1} (V_A - V_B)}, \tag{2}$$

where E^{-1} is the covariance matrix.

The correlation coefficient may be calculated both for the matrix and vector representations of a shape. The correlation between two matrices can be derived using the formula [16]:

$$c_c = \frac{\sum_m \sum_n (A_{nm} - \bar{A})(B_{nm} - \bar{B})}{\sqrt{\left(\sum_m \sum_n (A_{nm} - \bar{A})^2\right)\left(\sum_m \sum_n (B_{nm} - \bar{B})^2\right)}}, \tag{3}$$

where:
A_{mn}, B_{mn} — pixel value with coordinates (m,n), respectively in image A and B,
$\bar{A}$, $\bar{B}$ — average value of all pixels, respectively in image A and B.

The C1 metric is also a similarity measure based on shape correlation. It is obtained by means of the following formula [12]:

$$c_1(A,B) = 1 - \frac{\sum_{i=1}^{H}\sum_{j=1}^{W}\left|a_{ij}-b_{ij}\right|}{\sum_{i=1}^{H}\sum_{j=1}^{W}\left(\left|a_{ij}\right|-\left|b_{ij}\right|\right)} \tag{4}$$

where:
A, B — matched shape representations,
H, W — height and width of the representation.

3 Selected Shape Descriptors

3.1 Two-Dimensional Fourier Descriptor

The use of Fourier-based shape descriptors is widespread in pattern recognition and valued for its properties, including shape generalisation, robustness to noise, scale invariance and translation invariance. The Two-Dimensional Fourier Descriptor (2DFD) has the form of a matrix with absolute complex values, and is derived as follows [9]:

$$C(k,l) = \frac{1}{HW}\left|\sum_{h=1}^{H}\sum_{w=1}^{W} P(h,w)\cdot\exp^{\left(-i\frac{2\pi}{H}(k-1)(h-1)\right)}\cdot\ldots\right.$$
$$\left.\ldots\cdot\exp^{\left(-i\frac{2\pi}{W}(l-1)(w-1)\right)}\right|, \tag{5}$$

where:
H, W — height and width of the image in pixels,
k — sampling rate in vertical direction ($k \geq 1$ and $k \leq H$),
l — sampling rate in horizontal direction ($l \geq 1$ and $l \leq W$),
$C(k,l)$ — value of the coefficient of discrete Fourier transform in the coefficient matrix in k row and l column,
$P(h,w)$ — value in the image plane with coordinates h, w.

3.2 UNL-Fourier

The UNL-Fourier (UNL-F) descriptor is composed of the UNL (named after Universidade Nova de Lisboa) descriptor and Fourier transform. The UNL utilizes a complex representation of Cartesian coordinates for points and parametric curves in discrete manner [17]:

$$z(t) = \left(x_1 + t\left(x_2 - x_1\right)\right) + \mathrm{j}\left(y_1 + t\left(y_2 - y_1\right)\right), \quad t \in (0,1), \tag{6}$$

where $z_1 = x_1 + \mathrm{j}y_1$ and $z_2 = x_2 + \mathrm{j}y_2$ are complex numbers. In the next step, the centroid O is calculated [17]:

$$O = (O_x, O_y) = \left(\frac{1}{n}\sum_{i=1}^{n} x_i, \frac{1}{n}\sum_{i=1}^{n} y_i\right), \tag{7}$$

and the maximal Euclidean distance between contour points and centroid is found [17]:

$$M = \max_i\left\{\left\|z_i(t) - O\right\|\right\} \quad \forall i = 1\ldots n \quad t \in (0,1). \tag{8}$$

Based on the above formulations, a discrete version of the new coordinates is calculated as follows [17]:

$$U(z(t)) = \frac{\left\|\left(x_1 + t(x_2+x_1) - O_x\right) + \mathrm{j}\left(y_1 + t(y_2+y_1) - O_y\right)\right\|}{M} + \mathrm{j} \times \arctan\left(\frac{y_1 + t(y_2 - y_1) - O_y}{x_1 + t(x_2 - x_1) - O_x}\right). \tag{9}$$

Original pixel values are put into a square Cartesian matrix based on the new coordinates. This results in an image containing unfolded shape contour in polar coordinates, in which rows represent distances from the centroid and columns — the angles. As a result, the 2DFD can be applied.

3.3 Generic Fourier Descriptor

The Generic Fourier Descriptor (GFD) utilizes the transformation of a region shape to the polar coordinate system. It means that all pixel coordinates of an original image are transformed into polar coordinates. Next, the original pixel values are put to new coordinates on a rectangular Cartesian image, in which the row elements correspond to distances from the centroid and the columns to angles [13]. Again, the result is two-dimensional and the Fourier transform can be applied.

3.4 Point Distance Histogram

The Point Distance Histogram (PDH) is a shape descriptor that utilizes information about the shape contour. In order to derive a PDH representation, an origin of the polar transform of a contour is firstly selected, usually a centroid O. Polar coordinates are stored in two vectors — Θ^i for angles and P^i for radii [3]:

$$\Theta_i = \mathrm{a}\tan\left(\frac{y_i - O_y}{x_i - O_x}\right), \tag{10}$$

$$P_i = \sqrt{\left(x_i - O_x\right)^2 + \left(y_i - O_y\right)^2}. \tag{11}$$

In the next step, the values in Θ^i are converted to the nearest integers. Then the elements in Θ^i and P^i are rearranged with respect to the increasing values in Θ^i, and denoted as Θ^j, P^j. If there are any equal elements in Θ^j, then only the element with the highest value in P^j is left. Next, only the P^j vector is selected for further processing and denoted as P^k, where k =1,2,...m and $m \leq 360$. The elements of P^k vector are normalized and assigned to bins in the histogram (ρ_k to l_k) [3]:

$$l_k = \begin{cases} r, & \text{if} \quad \rho_k = 1 \\ \lfloor r\rho_k \rfloor, & \text{if} \quad \rho_k \neq 1 \end{cases} \tag{12}$$

where r is a previously determined number of bins. In the next step, the values in the histogram bins are normalized according to the highest one. Ultimately, the final histogram which represents a shape is obtained and can be written as the following function $h(l_k)$ [3]:

$$h(l_k) = \sum_{k=1}^{m} b(k, l_k), \tag{13}$$

where [3]:

$$b(k, l_k) = \begin{cases} 1, & \text{if} \quad k = l_k \\ 0, & \text{if} \quad k \neq l_k \end{cases} \tag{14}$$

3.5 Zernike Moments

Zernike Moments (ZM) are orthogonal moments. Among the advantages of this descriptor are rotation invariance, robustness to noise and minor variations in shape. The complex Zernike Moments are derived from orthogonal Zernike polynomials, which are a set of functions orthogonal over the unit disk $x^2 + y^2 < 1$. The Zernike Moments of order n and repetition m of a region shape $f(x,y)$ can be obtained by the following formula [13]:

$$Z_{nm} = \frac{n+1}{\pi} \sum_{r} \sum_{\theta} f(r\cos\theta, r\sin\theta) \cdot R_{nm}(r) \cdot \exp(jm\theta), \quad r \leq 1. \tag{15}$$

where $R_{nm}(r)$ is the orthogonal radial polynomial [13]:

$$R_{nm}(r) = \sum_{s=0}^{(n-|m|)/2} (-1)^s \cdot \ldots \cdot \frac{(n-s)!}{s! \times \left(\frac{n-2s+|m|}{2}\right)! \left(\frac{n-2s-|m|}{2}\right)!} r^{n-2s}, \tag{16}$$

where $n = 0, 1, 2, \ldots$; $0 \leq |m| \leq n$; $n - |m|$ is even.

4 The Description of the Experiments and Experimental Results

During the experiments, five different shape descriptors and four matching methods were used. In each experiment, one combination of a shape descriptor and matching method was investigated. Firstly, all shapes were represented using a selected variant of the shape descriptor — a feature vector, i.e. part of the absolute spectrum in case of 2DFD, GFD or UNL-F, various orders for ZM and various number of histogram bins for PDH. Next, the representations of test objects were matched with the representations of templates by calculating the similarity or dissimilarity measure. Lastly, one most similar template was selected for each investigated object, giving a set of templates.

At this point it is important to take a closer look at the data and shape representations. The shapes that were used in the experiments are depicted in Fig. 1 and consisted of 200×200 pixel size images with a white background and black silhouettes of similar size placed in the middle. The shapes consisted of ten shapes that were general templates (the first row in Fig. 1) and test objects. The shape representations varied significantly in terms of size. In case of shape descriptors based on the Fourier transform, various parts of the original absolute spectrum were investigated, namely 2×2, 5×5, 10×10, 25×25 and 50×50 subparts of the coefficient matrix. Each block was transformed into a vector to form a final shape representation. The Zernike Moments descriptor was calculated for orders from 1 to 20, what resulted in feature vectors having from 2 to 121 elements. The Point Distance Histogram descriptor had seven variants that were obtained for 2, 5, 10, 25, 50, 75 and 100 histogram bins, and simultaneously produced feature vectors of size equal to the number of bins.

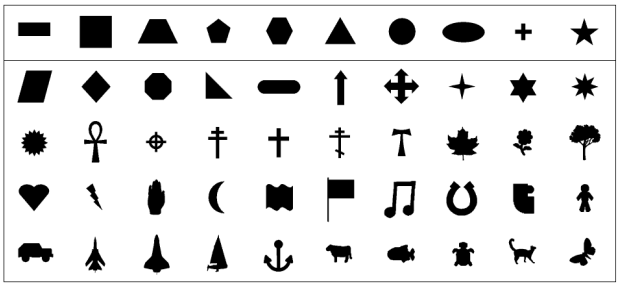


Figure 1: Shapes used in the experiments divided into 10 templates (first row) and 40 test objects (rest) [3].

The effectiveness of the experiment was estimated by calculating the percentage of the templates selected in the experiment that was consistent with the templates indicated by people in the inquiries concerning the same GSA task. In the inquiry people were asked to indicate five most similar templates for all test objects and arrange them from the most to the least similar one. In the paper, for an individual test object, only three out of five templates were taken into account and are compared with one template resulted from the experiments — here the sequence of indications is not taken into account. Templates indicated by people are provided in Fig. 2. The percentage differences of indications between the most and the second most similar templates are various and depend on the test object. For instance, for test object no. 14 a cross template was indicated by 65% of people and the star template by 64%. In case of test object no. 4 the difference was greater — 94% of people indicated a triangle and 46% indicated trapeze.

The aim of the experiments was to select the combination of a shape descriptor and matching method that gave the highest effectiveness and, additionally, in the case of several combinations with the same percentage effectiveness, in which the size of the shape representation would be the smallest. The following part of this section describes the experimental results.

The first set of the experiments utilized the Two-

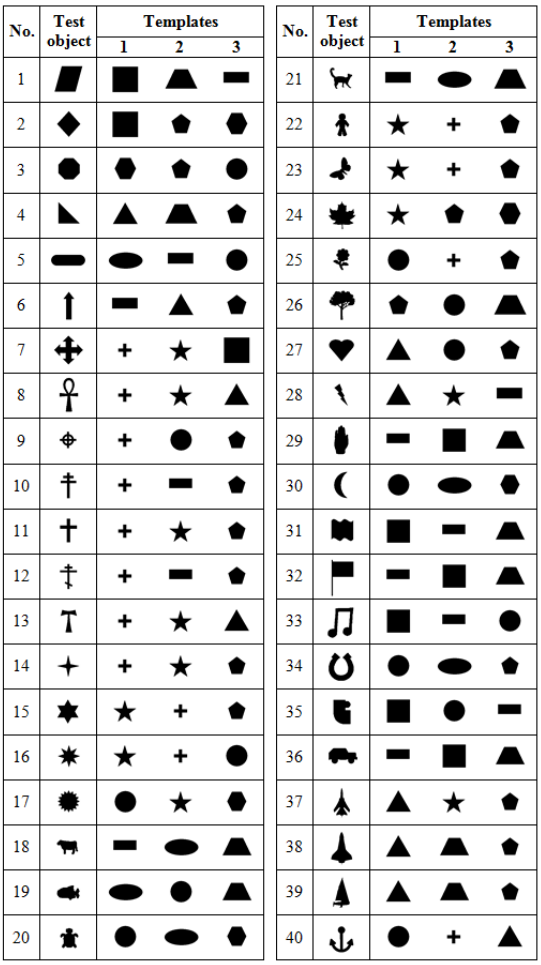


Figure 2: Templates most frequently indicated by people in the inquiries.

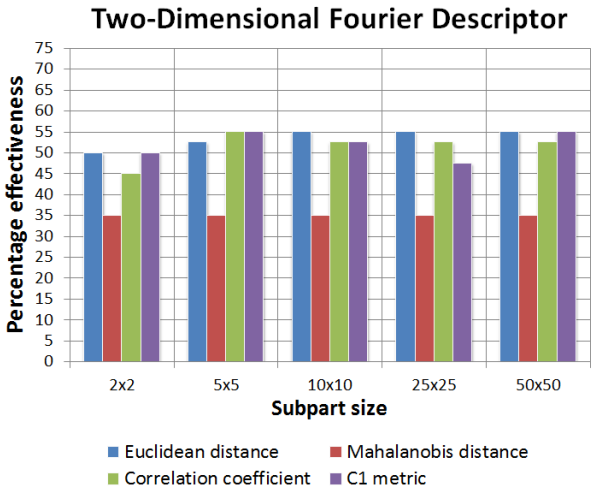


Figure 3: Bar chart representing the experiment results using the 2DFD.

Dimensional Fourier Descriptor and five different absolute spectrum subparts. The percentage effectiveness values for each combination of a shape descriptor and matching method are provided in Fig. 3. As can be seen in Fig. 3, the effectiveness values vary significantly and the weakest results were achieved in case of the use of the Mahalanobis distance. The highest effectiveness was obtained in the case of combinations with the percentage value equal to 55%. The best result can be attributed to the 5×5 subpart of the 2DFD and both similarity measures — correlation coefficient and C1 metric.

In the second set of the experiments, the Generic Fourier Descriptor was used and again five absolute spectrum subparts were investigated (see Fig. 4). Compared to the previous experiment, the best result was obtained using a dissimilarity measure — the Euclidean distance, and the smallest feature vector — 2×2 subpart of the absolute spectrum. Similarly as in the previous case, the Mahalanobis distance provided the lowest effectiveness values.

The third set of the experiments included the application of the UNL-Fourier descriptor and again various subparts

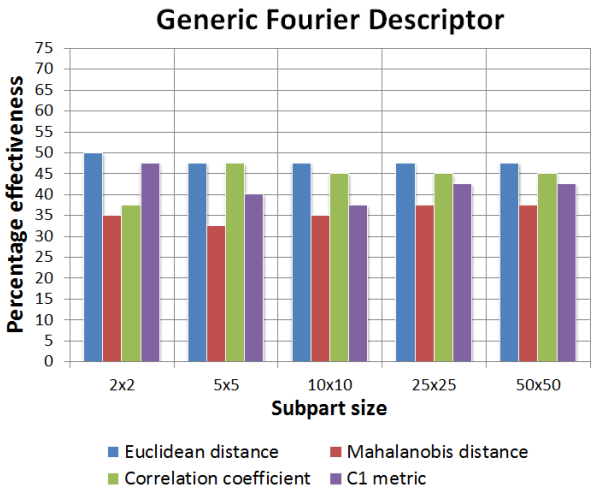


Figure 4: Bar chart representing the experiment results using the GFD.

of the Fourier coefficient matrix. The results are provided in Fig. 5. Three combinations stood out — 2×2 and 5×5 subparts of the UNL-F, which were matched using Euclidean distance, and 2×2 subpart of the UNL-F matched using C1 metric. These combinations gave 62,5% twice and 70% respectively. It is worth noting that the smallest feature vectors were sufficient to distinguish templates from each other and indicate the templates consistent with human indications.

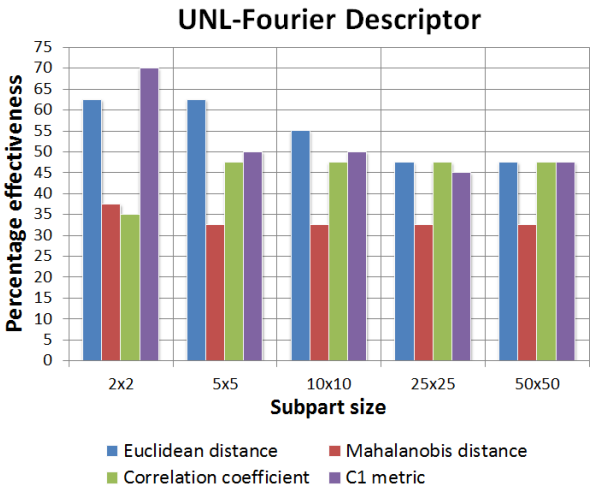


Figure 5: Bar chart representing the experiment results using the UNL-F.

The fourth set of experiments concerned the investigation of the effectiveness of Zernike Moments descriptor and different orders of moment were used (see Fig. 6). The results are varied — the percentage effectiveness values range from 22.5% to 60%. Suprisingly, the best results were observed when the Mahalanobis distance was appied as the matching method and the first-order moment was used. In this case the feature vector had only two elements.

The last set of the experiments examined the Point Distance Histogram descriptor. A different number of histogram bins was utilized, what resulted in a varying number of elements in each feature vector. As can be seen in Fig. 7, the highest effectiveness value was equal to 50% and was obtained for the combination of the PDH descriptor calculated for five histogram bins and C1 metric.

5 Conclusions

The paper covered the problem of the General Shape Analysis and investigated some solutions to it. Firstly, the idea underlying the approach was introduced and its possible applications, as well as several methods and algorithms that are already in use were briefly presented. In solving the GSA problem we are establishing the degree of similarity between test objects and general templates — one or few templates, which are most similar to an inves-

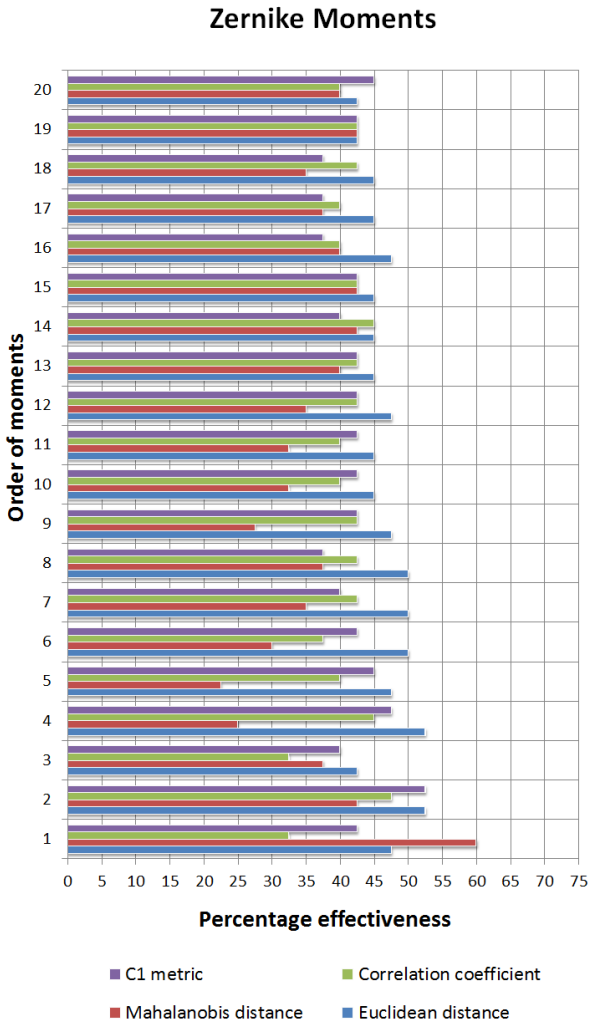


Figure 6: Bar chart representing the experiment results using the ZM.

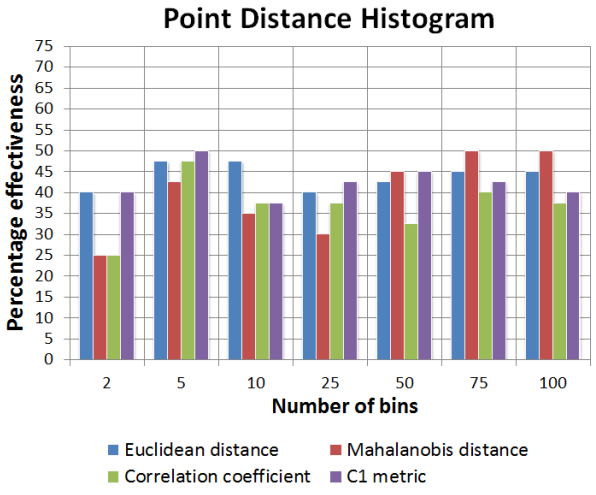


Figure 7: Bar chart representing the experiment results using the PDH.

tigated object are selected and compared with benchmark results in order to estimate the effectiveness of the experiment. The main goal of the experiments presented in the paper was to examine various combinations of shape descriptors and matching methods. Five shape descriptors were used to calculate shape representations (feature vectors) of various size. The descriptors comprised the Two-Dimensional Fourier Descriptor, Generic Fourier Descriptor, UNL-Fourier, Zernike Moments and Point Distance Histogram. The matching methods included two similarity measures, namely the correlation coefficient and C1 metric, and two dissimilarity measures — the Euclidean and Mahalanobis distances. Based on the experimental results, the best solution for the GSA problem was selected, i.e. a combination of a shape descriptor and matching method, which gave the highest percentage effectiveness. What is more, the smaller the feature vector the better the result. On the basis of the abovementioned criteria, the best solution for the GSA problem is the combination of the UNL-F descriptor, 2×2 subpart of the absolute spectrum and C1 metric. Pictorial results are provided in Fig. 8. Additionally, both the calculation of description vectors (shapes and templates together) and similarity measures between shapes are not time-consuming. There are slight differences between runtimes when using various matching methods and previously calculated descriptors (see Fig. 9), however they are not significant for small-sized description vectors.

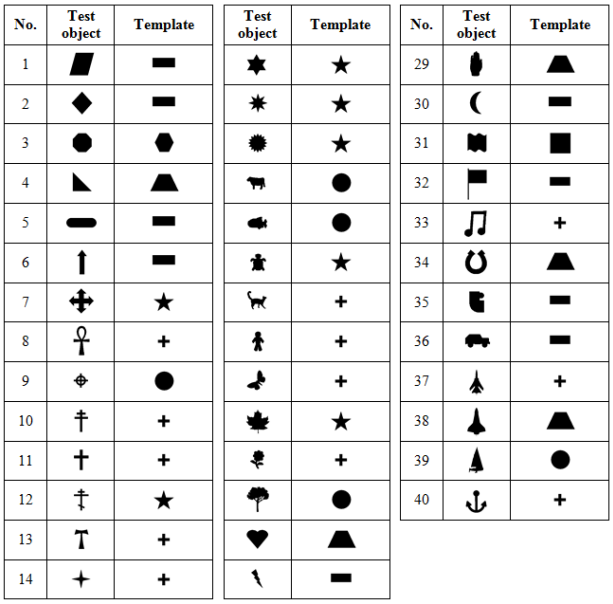


Figure 8: Results of the best experiment using UNL-Fourier descriptor and C1 metric.

By way of conclusion, it needs to be highlighted that the matching method has a significant impact on the final effectiveness of the experiment. Moreover, the effectiveness values also depend on the applied version of the shape descriptor. In other words, taking into consideration solely

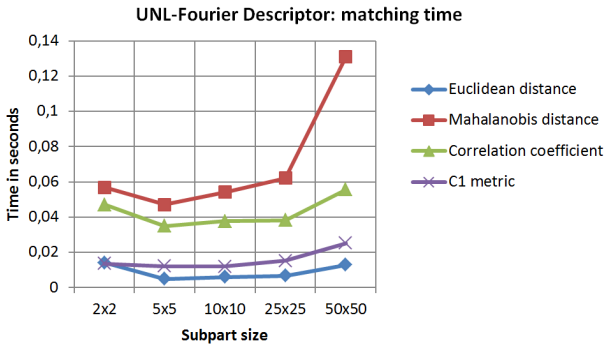


Figure 9: A comparison of matching times for various size of UNL-F Descriptor and matching methods.

one particular shape description algorithm, each combination of a feature vector and matching method produces different experimental results. This in turn may indicate that some feature vectors represent significant shape features in a more appropriate way, enabling easy recognition and matching of all shapes with common general characteristics. However, a matching method does not change the original efficiency of the shape description algorithm. A high diversity in effectiveness values stems from the fact that each matching method is based on different inputs, therefore it should be properly selected to fit the actual problem and the shape descriptor applied. Summarizing, three factors can affect the final experimental result: a shape description algorithm, the size of a feature vector and a method for estimating similarity between shape representations.

References

[1] Frejlichowski D. General shape analysis using fourier shape descriptors. In Swiatek J., Borzemski L., Grzech A., and Wilimowska Z., editors, *Information Systems Architecture and Technology — System Analysis in Decision Aided Problems*, pages 143–154. Oficyna Wydawnicza Politechniki Wrocławskiej, 2009.

[2] Frejlichowski D. An experimental comparison of seven shape descriptors in the general shape analysis problem. In Campilho A. C. and Kamel M. S., editors, *ICIAR (1)*, volume 6111 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 2010.

[3] Frejlichowski D. An experimental comparison of three polar shape descriptors in the general shape analysis problem. In Swiatek J., Borzemski L., Grzech A., and Wilimowska Z., editors, *Information Systems Architecture and Technology — System Analysis in Decision Aided Problems*, pages 139–150. Oficyna Wydawnicza Politechniki Wrocławskiej, 2010.

[4] Frejlichowski D. Application of zernike moments to the problem of general shape analysis. *Control and Cybernetics*, vol. 40, no. 2:515–526, 2011.

[5] Frejlichowski D. Application of the curvature scale space descriptor to the problem of general shape analysis. *Przeglad Elektrotechniczny (Electrical Review)*, no. 10b/2012:209–212, 2012.

[6] Frejlichowski D. and Gościewska K. Application of 2d fourier descriptors and similarity measures to the general shape analysis problem. In Bolc L., Tadeusiewicz R., Chmielewski L. J., and Wojciechowski K. W., editors, *ICCVG*, volume 7594 of *Lecture Notes in Computer Science*, pages 371–378. Springer, 2012.

[7] Zhang D. *Image Retrieval Based on Shape*. Dissertation, Faculty of Information Technology, Monash University, Australia, 2002.

[8] Luciano da Fontoura Costa and Roberto Marcond Cesar Jr. *Shape Analysis and Classification: Theory and Practice*. CRC Press, 2000.

[9] Kukharev G. *Digital Image Processing and Analysis (in Polish)*. SUT Press, 1998.

[10] Biederman I. Recognition-by-components: A theory of human image understanding. *Psychological Review*, vol. 94, no. 2:115–147, 1987.

[11] Kpalma K. and Ronsin J. An overview of advances of pattern recognition systems in computer vision. In Obinata G. and Dutta A., editors, *Vision Systems: Segmentation and Pattern Recognition*. I-Tech, Vienna, Austria, 2007.

[12] Lam K.-M. and Yan H. An analytic-to-holistic approach for face recognition based on a single frontal view. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(7):673–686, Jul 1998.

[13] Yang M., Kpalma K., and Ronsin J. A survey of shape feature extraction techniques. In Peng-Yeng Yin, editor, *Pattern Recognition*, pages 43–90. I-Tech, Vienna, Austria, 2008.

[14] Forczmański P. and Frejlichowski D. Robust stamps detection and classification by means of general shape analysis. In Bolc L., Tadeusiewicz R., Chmielewski L. J., and Wojciechowski K. W., editors, *ICCVG (1)*, volume 6374 of *Lecture Notes in Computer Science*, pages 360–367. Springer, 2010.

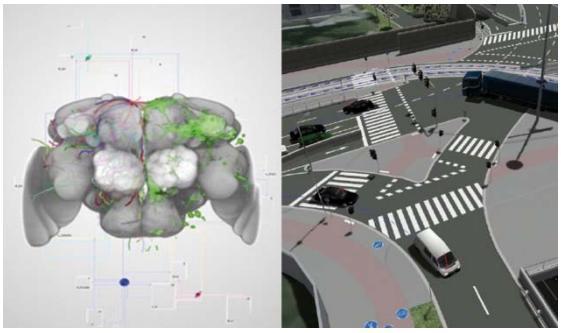
[15] Youguo Pi, Wenzhi Liao, Mingyou Liu, and Jianping Lu. Theory of cognitive pattern recognition. In Peng-Yeng Yin, editor, *Pattern Recognition Techniques, Technology and Applications*. I-Tech, Vienna, Austria, 2008.

[16] Chwastek T. and Mikrut S. The problem of automatic measurement of fiducial mark on air images (in polish). *Archives of Photogrammetry, Cartography and Remote Sensing*, 16:125–133, 2006.

[17] Rauber T. W. Two dimensional shape description. Technical report, Universidade Nova de Lisboa, Lisoba, Portugal, 1994.

Sponsors of CESCG 2014

zentrum für
virtual reality und visualisierung
forschungs-gmbh



VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH

The VRVis Research Center is a joint venture in research and development for virtual reality and visualization. VRVis was founded in 2000 as part of the Austrian Kplus program to bridge the gap between academic research and commercial development as well as to supply the necessary transfer of knowledge between the academic community and industry. The competence center VRVis is funded by BMVIT, BMWFW, and ZIT - The Technology Agency of the City of Vienna within the scope of COMET – Competence Centers for Excellent Technologies. The program COMET is managed by FFG.

This mission is mirrored in a variety of academic and industrial partners. The research center is currently conducted by five academic institutes and numerous industrial partners. Leading-edge innovations and down-to-earth business style characterizes VRVis as a valued partner for high-level research.

The company is located in Vienna, Austria. Today, around 60 researchers together with about 20 students do high-level applied and basic research in three different areas.

The Team

VRVis consists of internationally experienced researchers in the areas of visualization, rendering and visual analysis. Their outstanding experience and knowledge in these topics qualify them for the innovative research they are performing. The research areas are headed by key researchers who manage these areas, define goals and projects for this area, and conduct the defined research together with their staff. All members of the research team are young researchers, whose creativity and ingenuity is the key to the success. VRVis is always looking for young, talented, and motivated researches in the fields of research to extend its research work or to support partner companies.

Research Program

The scientific research program consists of three research areas (Visualization, Rendering and Visual Analysis) in which thematically matching research projects are conducted. Each research area realizes application projects on the one hand and basic research for these application projects on the other hand.

Working at VRVis

VRVis is always looking for students, junior and senior researchers who want to join the VRVis team. VRVis is offering internships, diploma theses and PhD theses in cooperation with the TU Wien and regular positions. For more information or search for job opportunities in the field of Visual Computing visit our webpage at www.vrvis.at.

Selection of Partners

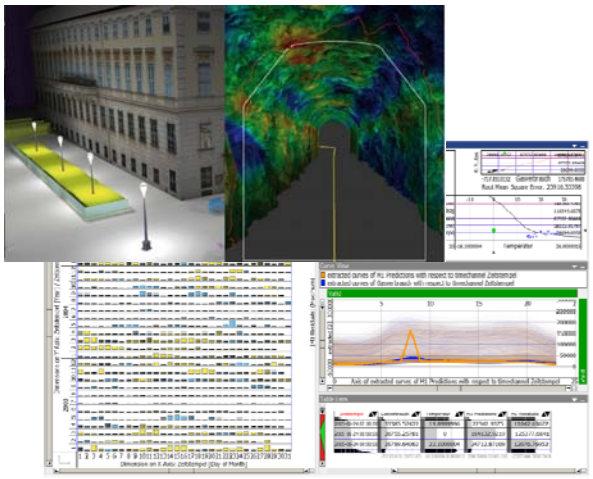
Scientific Partners:

- Vienna University of Technology
- Graz University of Technology

Industrial Partners:

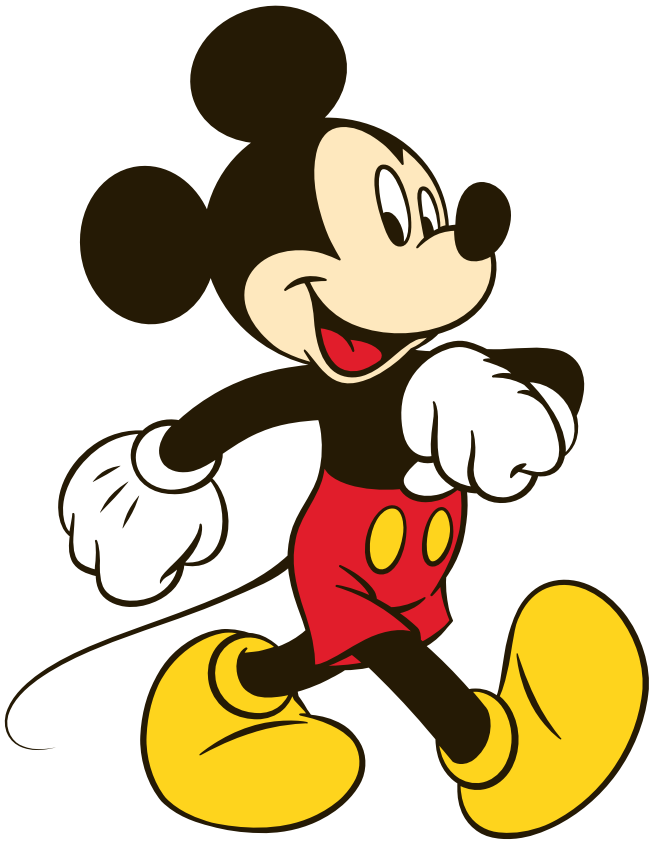
- AVL List GmbH
- AGFA Healthcare GesmbH
- Austria Power Grid AG
- Geodata Ziviltechniker GmbH
- Imagination Computer Services GesmbH
- ÖBB-Infrastruktur AG
- Zumtobel Lighting GmbH
- and many more

Currently, VRVis is again extending its industrial base with new partners from several new fields.



Additional Information and Contact

Please visit our webpage for detailed information about the research program or current projects at www.vrvis.at or contact us at office@vrvis.at or via phone +43 (1) 20501 / 30100.



Disney Research

As part of The Walt Disney Company, Disney Research draws on a legacy of innovation and technology leadership that continues to this day. In 1923, Walt Disney sold his animated/live-action series the Alice Comedies, founded his eponymous company, and launched a succession of firsts: The first cartoon with fully synchronized sound (1928). The first full-color cartoon (1932). The first animated feature film (1937). The first modern theme park (1955).

The Walt Disney Company was also an early leader in entertainment technology development with inventions like the multiplane camera, Audio-Animatronics, Circle-Vision 360°, and Fantasound.

In 2006, The Walt Disney Company acquired Pixar Animation Studios, a move that brought a host of valuable creative and technology assets, including a strong culture of excellence in research. Pixar is a major generator and publisher of world-class research in computer graphics. Its scientists contribute directly to Pixar's critically acclaimed films, consistently winning multiple technical Academy Awards®. The Pixar acquisition was a source of inspiration for the formation of Disney Research, and continues to influence the way we're organized and run.

Disney Research was launched in 2008 as an informal network of research labs that collaborate closely with academic institutions such as Carnegie Mellon University and the Swiss Federal Institute of Technology Zürich (ETH). We're able to combine the best of academia and industry: we work on a broad range of commercially important challenges, we view publication as a principal mechanism for quality control, we encourage engagement with the global research community, and our research has applications that are experienced by millions of people. We're honoring Walt Disney's legacy of innovation by researching novel technologies and deploying them on a global scale.