REAL-TIME NON-PHOTOREALISTIC RENDERING TECHNIQUES FOR ILLUSTRATING 3D SCENES AND THEIR DYNAMICS

DISSERTATION
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
DOCTOR RERUM NATURALIUM
(DR. RER. NAT.)
IN DER WISSENSCHAFTSDISZIPLIN INFORMATIK

EINGEREICHT AN DER
MATHEMATISCH-NATURWISSENSCHAFTLICHEN FAKULTÄT
DER UNIVERSITÄT POTSDAM

VON
MARC NIENHAUS
GEBOREN AM 2.9.1973 IN RHEDE

POTSDAM, DEN 20. JUNI 2005

ABSTRACT

This thesis addresses real-time non-photorealistic rendering techniques and their applications in interactive visualization. *Real-time rendering* has emerged as an important discipline within computer graphics developing a broad variety of rendering and optimization techniques along with dramatic advances in computer graphics hardware. While many applications of real-time rendering techniques concentrate on achieving photorealistic imagery, *non-photorealistic computer graphics* is investigating concepts and techniques that deliberately abstract from reality using expressive, stylized, or illustrative rendering; major goals include visual clarity, attractiveness, comprehensibility, and perceptibility in depictions. Non-photorealistic rendering techniques often rely on the concepts and principles found in traditional illustrations, graphics design, and art.

The contributions of this thesis include three general-purpose real-time non-photorealistic rendering techniques:

- The *edge-enhancement rendering technique* accentuates visually important edges of 3D models facilitating the effective communication of their shape. The technique takes an image-space approach for edge detection and encodes the resulting edge intensities as texture, called edge map, to enhance 3D models on a per-object basis.
- The *blueprint rendering technique* extends the edge-enhancement technique to the 3D models' occluded parts to accentuate their visible as well as their occluded visually important edges. Vivid and expressive depictions of complex aggregate objects become possible that facilitate the visual perception of spatial relationships and let viewers obtain insights into the models.
- The *sketchy drawing rendering technique* stylizes visually important edges of 3D models. Depicting 3D models in a sketchy manner allows us to express vagueness and is vitally important for communicating ideas and for presenting a preliminary, incomplete state.

Two applications based on these real-time non-photorealistic rendering techniques in the fields of visualization demonstrate their ability to build compelling, interactive visual interfaces:

- *Illustrative 3D city models* apply non-photorealism to represent virtual spatial 3D environments together with associated thematic information. The abstracted, stylized depiction emphasizes components of 3D city models and thereby eases recognition, facilitates navigation, exploration, and analysis of spatial information.
- *Illustrative CSG models* apply non-photorealism to image-based CSG rendering. They enable us to visualize the design and assembly of complex CSG models in a comprehensible fashion. It also simplifies the interactive construction of CSG models.

Finally, the thesis investigates an automated approach to depict dynamics as a complementary, important dimension in information contents by means of non-photorealistic rendering:

- The *smart depiction system* automatically generates compelling images of a 3D scene's related dynamics following the traditional design principles found in comic books and storyboards. The system symbolizes past, ongoing, and future activities and events taking place in and related to 3D scenes.

The non-photorealistic rendering techniques and exemplary applications presented in this thesis demonstrate that non-photorealistic rendering serves as a fundamental technology for expressive and effective visual communication and facilitates the implementation of user interfaces based on illustrating 3D scenes and their related dynamics in an informative and comprehensible way.

ZUSAMMENFASSUNG

Diese Arbeit behandelt echtzeitfähige nichtphotorealistische 3D-Renderingverfahren und darauf basierende Anwendungen der interaktiven Visualisierung.

Das *Echtzeit-Rendering* hat sich als eine bedeutende Wissenschaftsdisziplin innerhalb der Computergraphik etabliert, die in Verbindung mit dem rasanten Fortschritt in der Computergraphikhardware vielfältige Rendering- und Optimierungsverfahren hervorgebracht hat. Während sich viele computergraphische Anwendungen auf die Erzeugung photorealistischer Bilder konzentrieren, erforscht das Gebiet der *nichtphotorealistischen Computergraphik* Konzepte und Prinzipien, die durch den Einsatz von expressiven und illustrativen Darstellungstechniken bewusst vom Realismus abweichen. Gegenstand des nichtphotorealistischen 3D-Renderings ist die Erzeugung von verständlichen, attraktiven und aussagekräftigen Abbildungen von 3D-Szenen. Um ausdrucksstarke Bildinhalte zu generieren beruhen nichtphotorealistische Renderingverfahren oftmals auf den Erfahrungen und Leitlinien der traditionellen Illustration, des Graphikdesigns und der Kunst.

Der Beitrag dieser Arbeit beinhaltet drei universell nutzbare echtzeitfähige Renderingverfahren:

- Das *Renderingverfahren zum Hervorheben von visuell bedeutsamen Kanten* von 3D-Modellen ermöglicht eine effektive Darstellung der geometrischen Form dieser Modelle. Das Verfahren beruht auf einem bildbasierten Ansatz zur Kantendetektion und speichert die daraus resultierenden Intensitätswerte in einer Textur, der sogenannten *Edge Map*, um anschließend die Kanten pro Objekt individuell zu betonen.
- Das *Blueprint Renderingverfahren* wendet das Verfahren zum Hervorheben von Kanten auf die verdeckten Bestandteile eines 3D-Modells an. Auf diese Weise können sowohl die sichtbaren als auch die verdeckten visuell bedeutsamen Kanten eines 3D-Modells graphisch hervorgehoben werden. Das Verfahren ermöglicht anschauliche und ausdrucksstarke Darstellungen von komplex aggregierten Objekten, die deren räumlichen Zusammenhänge visuell erfassen lassen und somit eine verständliche Einsicht in deren Aufbau bieten.
- Das *Renderingverfahren zur skizzenhaften Darstellung* von 3D-Modellen stilisiert die visuell bedeutsamen Kanten. Skizzenhafte Darstellungen können Unklarheit und Unsicherheit visualisieren und sind daher zur Kommunikation von Ideen und zur Präsentation von vorläufigen oder unvollendeten Entwürfen besonders geeignet.

Zwei Anwendungen auf dem Gebiet der Visualisierung nutzen die vorgestellten echtzeitfähigen nichtphotorealistischen Renderingverfahren und veranschaulichen deren Leistungsfähigkeit zur Generierung überzeugender interaktiver visueller Schnittstellen:

- *Illustrative 3D-Stadtmodelle* visualisieren virtuelle räumliche Umgebungen und der mit diesen verknüpften thematischen Informationen. Die abstrahierten und stilisierten Darstellungen heben die Komponenten der 3D-Stadtmodelle besonders hervor, erleichtern somit deren Identifikation und ermöglichen eine effiziente Navigation, Exploration und Analyse der räumlichen Informationen.
- *Illustrative CSG-Modelle* kombinieren Nichtphotorealismus mit bildbasiertem CSG-Rendering. Die illustrativen Darstellungen visualisieren Aufbau und Anordnung von komplexen CSG-Modellen in einer verständlichen Art und Weise und vereinfachen somit deren interaktive Konstruktion.

Schließlich untersucht diese Arbeit einen auf dem Nichtphotorealismus basierenden Ansatz zur automatischen, bildhaften Darstellung von Dynamik, welche eine komplementäre und bedeutende Dimension von Informationsinhalten in 3D-Szenen repräsentiert:

- Das *System zur bildhaften Darstellung von Dynamik* erzeugt verständliche Abbildungen der Dynamik in 3D-Szenen automatisch unter Berücksichtigung von Entwurfsrichtlinien, die traditionell zum Zeichen von Comics und Storyboards genutzt werden. Das System symbolisiert dazu die in einer 3D Szene vergangenen, andauernden und zukünftigen Abläufe und Ereignisse.

Die in dieser Arbeit vorgestellten nichtphotorealistischen Renderingverfahren und deren Anwendungen in der interaktiven Visualisierung verdeutlichen, dass nichtphotorealistisches Rendering eine grundlegende Methodik zur expressiven und effektiven visuellen Kommunikation ist und, basierend auf informativen und verständlichen Illustrationen von 3D-Szenen und deren Dynamik, die Entwicklung von neuen graphischen Schnittstellen fördert.

ACKNOWLEDGEMENTS

This dissertation is the result of my research work at the Department of Computer Graphics Systems at the Hasso-Plattner-Institute (University of Potsdam). I am very grateful to Prof. Dr. Jürgen Döllner, my adviser, for giving me this opportunity. I am also extremely grateful to him for many enlightening comments, for "raising the bar" steadily, and for supporting me pursue my own research ideas. His experience in the field, enthusiasm for research and writing, and his demand for "good" and visually compelling depictions are continuous stimuli for my own work.

I would also like to thank my external reviewers Prof. Dr. Klaus Hinrichs from the University of Münster, who originally raised my interests in computer graphics, Prof. Dr. Thomas Strothotte from the University of Magdeburg, and Prof. Dr. Klaus Rebensburg from the University of Potsdam for unhesitatingly taking on the review of this thesis.

I am very grateful to Florin Kirsch, my former colleague for collaborating on combining blueprint rendering with image-based CSG rendering and on local multipass rendering methods, for maintaining the *Virtual Rendering System*, and for fruitful discussions on computer graphics hardware. I am also very grateful to Henrik Buchholz, my colleague, for collaborating on *Illustrative 3D City Models,* for sharing an office, and for the fun that we have. And I am also very grateful to Penny Salter for proofreading this thesis and *Sketchy Drawings*, and for rehearsing my *Graphics Interface* talk.

Thanks are also due to Oliver Kersting, my former colleague for tips on "how to get a thesis started and probably finished successfully", Dr. Konstantin Baumann for steadily improving the Virtual Rendering System using efficient implementations, and Sabine Biewendt, Stefan Maass, Johannes Bohnet, and Haik Lorenz for enriching the computer graphics group.

I would also wish to express my sincere gratitude to:
Amy and Bruce Gooch for encouraging the work on *Sketchy Drawings* and for providing me with the model of the crank that I used for illustrations in papers and throughout this thesis.
Prof. Ulrich Weinberg, from the Academy of Film and Television "Konrad Wolf" in Babelsberg, for supporting the project *3D-Storboards*, for inspiring discussions and sharing his knowledge of film making, and for the great trip to the Game Developers Conference in San Jose, to the University of Berkeley, and to the Stanford University. Andreas Puskeiler for helpful comments on design principles used in storyboards and for valuable references.
Cem Cebenoyan and Matt Pharr (NVIDIA Corporation) for reviewing *Blueprint Rendering and 'Sketchy Drawings'* and for publishing it in *GPU Gems II*. Christopher Keane for copy-editing the book contribution and Tammi Titsworth for copy-editing *Depicting Dynamics using Principles of Visual Art and Visual Narrations*.
Anonymous reviewers for presented suggestions on papers.

Last but certainly not least, I am very grateful to Ursel and Ludger Nienhaus, my parents, Julia, my sister, and Tobias, my brother, for their love and for a wonderful life, and for lively support, i.e., by continuously asking: "When will you finish your thesis?"
I am extremely grateful to Astrid Glösekötter, my girlfriend, for her love, for supporting me in frustrating moments during my research period, and, most importantly, for encouraging me to start my research work at all. In the end, she's the person who has made this thesis happened. Thanks.

CONTENTS

ABSTRACT iii

ZUSAMMENFASSUNG v

ACKNOWLEDGEMENTS vii

CONTENTS ix

INTRODUCTION 1

NON-PHOTOREALISTIC RENDERING 5
2.1 Determining and Stylizing Silhouettes 6
2.2 Applications of Non-Photorealistic Rendering 9

HARDWARE-ACCELERATED COMPUTER GRAPHICS 13
3.1 Programmable Rendering Pipeline 13
3.2 The OpenGL Shading Language 16
3.3 Render-to-Texture 18
3.4 Image Processing on Graphics Hardware 19
3.5 Dependent Texture Access 21
3.6 Deferred Shading 21
3.7 Depth Sprite Rendering 22

REAL-TIME EDGE-ENHANCEMENT RENDERING ALGORITHM 25
4.1 G-Buffer Concept 26
4.2 Edge Classification 28
4.3 Edge Map Construction 29
4.4 Applying the Edge Map 33
4.5 Intermediate Rendering Passes 33
4.6 Applications of Real-Time Edge-Enhancement Rendering 34
4.7 Conclusions 36
4.8 Application to Illustrative 3D City Models 40

BLUEPRINT RENDERING TECHNIQUE 45
5.1 Depth Peeling Technique 46
5.2 Blueprint Rendering 49
5.3 Edge-Enhanced Order-Independent Transparency 51

5.4 Depth-Masking ... 52
5.5 Applications to Illustrations of Ancient Architecture ... 54
5.6 Applications to Illustrations of CSG Models ... 55
5.7 Conclusions ... 58

"SKETCHY DRAWINGS" ... 61
6.1 Edges and Color Patches ... 62
6.2 Perlin Noise ... 63
6.3 Applying Uncertainty ... 63
6.4 Adjusting Depth ... 64
6.5 Drafts of Architecture ... 67
6.6 Variations of Sketchy Drawings ... 67
6.7 Controlling Uncertainty ... 68
6.8 Sketching Blueprints ... 70
6.9 Conclusions ... 71

DEPICTING DYNAMICS ... 73
7.1 Introduction to Depictions of Dynamics ... 75
7.2 Dynamics Glyphs ... 78
7.3 Specifying Scenes and Their Dynamics ... 78
7.4 Assembling Dynamics Information ... 81
7.5 Characterizing Dynamics ... 82
7.6 Symbolization and Depiction Techniques ... 83
7.7 Interactive Composition of Depictions ... 84
7.8 System Overview and User Involvement ... 85
7.9 Using Semantics for Depictions ... 86
7.10 Pattern-Based Symbolization ... 89
7.11 Conclusions and Future Work ... 91

SUMMARY AND CONCLUSIONS ... 93

BIBLIOGRAPHY ... 95

Chapter 1
Introduction

"Depiction is essentially an optimization problem, producing the best picture given goals and constraints" [32]. This statement from DURAND points out that a depiction has certain intentions besides just portraying scenes. These intentions are typically specified, for instance, by cognitive goals (i.e., ease of understanding depicted information), affective goals (i.e., invoking emotions), or motivational goals (i.e., motivating to participate in depicted concepts) [115], and, in addition, by the context a depiction is used in. Constraints are set by the limitations inherent to the 2D medium used, for example in the case of flat images, low-resolution images, and static images. The goal produces further constraints. For example, if a depiction needs to communicate a message, clarity of representation is essential.

The way a viewer perceives depictions depends also on her or his cultural, social [86], or professional background. Assessing and optimizing the quality of depictions requires principles of disciplines not only from graphics but also from cognitive science, perceptual psychology, cultural science, and visual arts.

Conveying Information Effectively

Choosing an optimal graphical form to achieve a specific goal represents always a challenging task. The graphical form has a significant impact on how a viewer interprets and understands a depiction.

Photorealistic depictions are based on techniques "that resemble the output of a photographic camera and that even make use of the physical laws being involved in the process of photography" [116]. But, a photorealistic depiction is not always the optimal choice for presenting visual information. "A simplified, abstract diagram is often preferred when an image is required to delineate and explain" [101]. LANSDOWN AND SCHOFIELD, for instance, raise the question in the context of maintenance manuals: "How much use is a photograph to mechanics when they already have the real thing in front of them?" [70].

Artists and graphic designers have found principles and techniques for generating pictures and diagrams to efficiently visualize information yet for a long time. TUFTE, for instance, describes the design strategy of the smallest effective difference, that is, "making visual distinctions as subtle as possible, but still clear and effective" [120]. In particular, "presenting a unified single idea with nothing complicated, extraneous, or contradictory in its makeup" [119] leads to efficient visual communication.

The field of non-photorealistic computer graphics is concerned with the science and technology that deliberately abstract from reality using expressive, stylistic, or illustrative rendering techniques. These generate images "that, generally speaking, appear to be made in part 'by hand'" [116]. Non-photorealistic rendering techniques can therefore provide new approaches to computer-generated visualization and exploration based on the concepts of visual art and cognitive science to address the human beings' ability to process visual information.

Take Advantage of Graphics Technology and Hardware

The ability of non-photorealistic rendering techniques to operate in real-time is vital for all interactive applications. Real-time rendering [5] has emerged as an important discipline within computer graphics developing a broad variety of rendering techniques along with the dramatic advances in computer graphics hardware. Setting up the real-time rendering process is typically managed by software interfaces for graphics hardware (e.g., OpenGL [108]) or, at a higher-level, by graphics libraries, such as scene graph libraries (e.g., OpenInventor [113], Virtual Rendering System VRS [29]). The rendering process itself is accelerated by graphics hardware to a significant amount. Non-photorealistic approaches can take advantage of technology and hardware available for real-time rendering.

Many real-time rendering techniques deploy specific features of graphics hardware technology to achieve a (near) photorealistic appearance. Such real-time rendering techniques include texture mapping, environment mapping, standard and soft shadow mapping [67], and bump mapping [62]. These techniques exploit graphic hardware capabilities and take advantage of high-level shading languages (e.g., OpenGL Shading Language [104]), which allow developers to program the 3D rendering pipeline. Hardware capabilities and shading languages can be deployed in a similar way for implementing non-photorealistic rendering techniques to achieve illustrative and expressive depictions in real-time.

Illustrating 3D Scenes and Their Dynamics

This thesis aims at designing, implementing, and applying real-time non-photorealistic rendering techniques for illustrating 3D scenes and their related dynamics. One particular goal is to identify general-purpose rendering components for real-time non-photorealistic rendering systems. These components should be based on today's computer graphics hardware, operate on arbitrary 3D scene contents, and collaborate with each other.

A fundamental technique in non-photorealistic rendering is concerned with the enhancement of the outlines of 3D models because outlines, particularly edges where the visibility of the surface changes (a.k.a. silhouettes), are one of the strongest visual cues [69], important "for figure-to-background distinction" [57], and essential for recognizing "the essence of shape" [56]. Rendering just the 3D models' outlines can depict them in a way that enables viewers to imagine details they might otherwise miss.

A real-time rendering technique for edge enhancement, therefore, can build a base for and extend a large number of illustrative and expressive rendering techniques. Taking this technique as a starting point, this thesis investigates to what extend edge enhancement can be applied to illustrate transparent 3D models, such as in blueprints, to enable comprehensible insights into the models and to make their aggregation easy to perceive. This thesis further investigates whether these edges can be used to illustrate 3D models in a sketchy way to enable the presentation of ideas and to encourage a viewer's participation.

Another goal of this thesis involves two applications based on the developed real-time non-photorealistic rendering techniques in the fields of visualization to demonstrate their ability to build compelling, interactive visual interfaces. The contributions include illustrative 3D city models, which apply non-photorealism to represent virtual spatial 3D environments together with associated thematic information. The abstracted, stylized depiction aims at emphasizing components of 3D city models and thereby improving recognition, facilitates navigation, exploration, and analysis of spatial information. The contributions include further illustrative

CSG models, which apply non-photorealism to image-based CSG rendering visualizing the design and assembly of complex CSG models in a comprehensible fashion.

Finally, the thesis investigates an automated approach to depict dynamics as a complementary, important dimension in information contents by means of non-photorealistic rendering. For it, a smart depiction system is presented that automatically generates compelling images of a 3D scene's related dynamics following the traditional design principles found in comic books and storyboards. The system symbolizes past, ongoing, and future activities and events taking place in and related to 3D scenes.

The non-photorealistic rendering techniques and exemplary applications presented in this thesis demonstrate that non-photorealistic rendering serves as a fundamental technology for expressive and effective visual communication and facilitates the implementation of user interfaces based on illustrating 3D scenes and their related dynamics in an informative and comprehensible way.

Structure of the Dissertation

The remainder of this dissertation is structured as follows:

Chapter 2 outlines the concepts and principles of non-photorealistic rendering. It reviews existing rendering algorithms for enhancing and stylizing the silhouettes of 3D geometries. The chapter presents applications of non-photorealistic rendering that motivate and illustrate serious usages of non-photorealistic rendering, and finally introduces the term *smart depiction system*.

Chapter 3 briefly describes the *programmable rendering pipeline*, a fundamental concept available on today's graphics hardware. The chapter gives an overview of existing hardware-accelerated rendering techniques that represent ingredients for real-time non-photorealistic rendering algorithms relevant to this work.

Chapter 4 presents a novel *real-time edge-enhancement rendering technique* that enables a distinctive display of 3D models and 3D scenes to increase visual perception. The chapter gives a definition of the polygonal 3D geometries' visually important edges considered for edge enhancement. The image-space algorithm is able to enhance models of high geometric complexity and large 3D scenes in real time. In a case study, this chapter describes how the edge-enhancement algorithm is applied for synthesizing non-photorealistic depictions of general, large-scale 3D city models.

Chapter 5 presents a novel *real-time blueprint rendering technique* that illustrates the design and spatial assembly of 3D models and their aggregated components. The resulting depictions provide clear insights into 3D models and enable one to perceive them as a whole. The chapter presents as a typical application of blueprint rendering the visualization and exploration of the layout and the assembly of mechanical parts and architecture. Since the composition of complex CSG models is particularly difficult to perceive, an extension of blueprint rendering to CSG rendering is presented as well. In this way, the blueprint rendering technique becomes an effective tool that assists the interactive construction of CSG models.

Chapter 6 presents a novel *real-time sketchy drawing rendering technique* that generates sketchy depictions of 3D models. Sketchy drawing can imply imprecision and vagueness to communicate visual ideas and to depict the preliminary states of a draft or concept. For example, architects and graphic designers can use sketchy drawing to encourage discussion and participation or to "sell" their ideas to their clients. The chapter also presents a combination of sketchy drawing and blueprint rendering, because both the interior and exterior features of 3D models can be subject for sketchy drawing as well.

Chapter 7 presents a novel smart depiction system that automatically generates compelling depictions of dynamics based on the traditional visual art and graphics design principles found in comic books and storyboards. The system relies on non-photorealistic rendering algorithms to generate storyboard-like depictions of 3D scenes and their related dynamics automatically. The depiction system demonstrates that non-photorealistic rendering affords novel application areas for computer-generated depictions.

Chapter 8 gives conclusions by summarizing and reviewing the presented depiction strategies. Potential future research directions are outlined.

Chapter 2

Non-Photorealistic Rendering

The term *non-photorealistic* basically qualifies pictorial styles that “do not attempt to imitate photography and to reach optical accuracy” [32]. In computer graphics, *non-photorealistic rendering* (*NPR*) [44][116] is concerned with rendering algorithms and techniques for synthesizing non-photorealistic visual representations of given scene descriptions.

Non-photorealistic rendering is difficult to classify in a positive manner. It is typically characterized by rendering techniques producing pictorial styles that simulate the “hand made” renditions that have a long tradition in art, illustrations, science, etc. Artists and graphic designers have found a means of presenting and depicting visual information purposefully with respect to specific domains and contexts. These skills are often based on principles derived from perceptual psychology or cognitive science. EDWARDS, for instance, proposes five skills of drawing a perceived object: *perception of edges*, *perception of space*, *perception of relationships*, *perception of lights and shadow*, and *perception of the whole, or gestalt* [34]. Artists and graphic designers have developed a rich set of well-elaborated techniques for producing effective pictures and diagrams to visualize information comprehensibly [120], attract visually, or make animations more convincing [119][124]. They use, for instance, line drawings in art (such as pen-and-ink drawings [74] or black-and-white illustrations [13]), for scientific or operational illustrations (such as medical or technical illustrations, maintenance manuals, architectural drafts, and storyboards [8][60]), and in entertainment graphics (such as comics [81] and cartoon movies [119]).

Since the advent of NPR in the last decade [48][70][106][125][126], numerous rendering styles have been introduced. There are a wealth of techniques including stylized digital halftoning to simulate handcrafted depictions, such as *stippling* [24], *hatching* [127], or *engraving* [25][96] which convey illumination, curvature, texture, and tone in an image (see Figure 2-1). Furthermore, techniques exist for generating technical illustrations automatically [43][45] or for reproducing pencil or pen-and-ink drawings [125][126].

It has long been understood that just a “few good lines” [112] often suffice to encourage viewers to complete a picture themselves by imagining the details that are missing. SOUSA AND PRUSINKIEWICZ [112] as well as DECARLO ET AL. [19][20] introduce algorithms for synthesizing line drawings from 3D geometries by placing lines purposefully to “suggest contours” and so to convey gestalts (see Figure 2-2). In entertainment graphics, TEECE introduces *Sable* for animated film production [117]. Sable is a paint-stroke rendering system that renders lines as stylized brush strokes for outlining and separating painterly colors (see Figure 2-2). In general, it can be observed that rendering 3D geometries by their outlines facilitates or complements most non-photorealistic rendering styles.

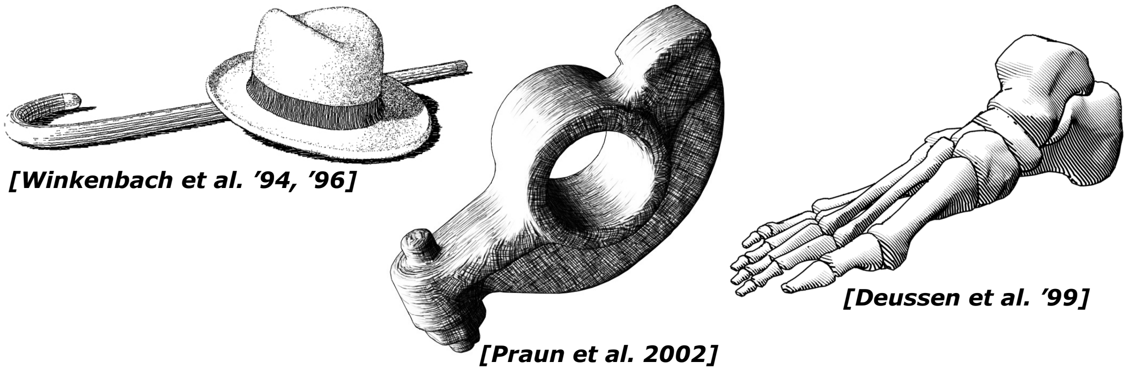


Figure 2-1: **Stylized Digital Halftoning.** Stylized digital halftoning provides visual cues in depictions, letting viewers perceive illumination, curvature, texture, and tone.

The remainder of this chapter is structured as follows: Section 2.1 gives an overview of well-established silhouette determination algorithms. Section 2.2 outlines the applications of non-photorealistic rendering that illustrate the diversity and the uses of non-photorealistic depictions and motivate subsequent work.

2.1 Determining and Stylizing Silhouettes

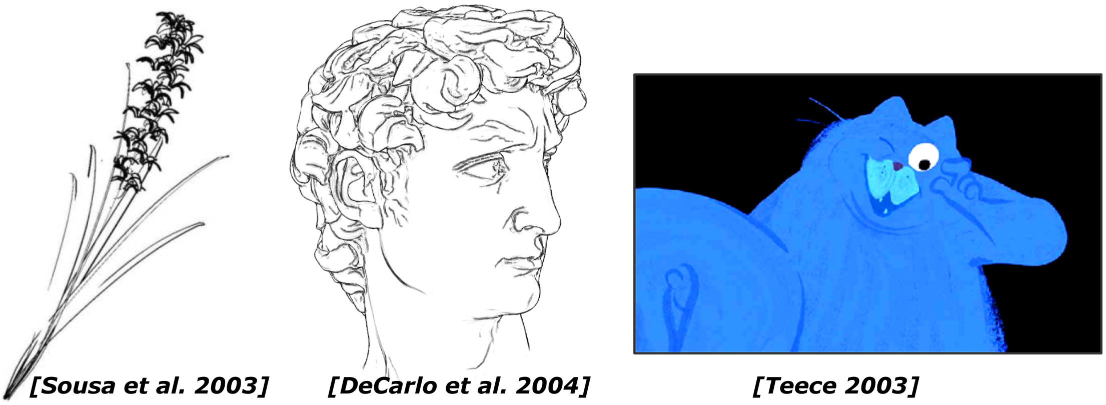
Determining and stylizing silhouettes represents a major research activity in non-photorealistic rendering. Developing efficient silhouette detection algorithms, typically using visibility determination techniques, is crucial, because silhouettes are particularly view dependent and so need to be detected for each single frame. Isenberg et al. [57] provide an exhaustive survey of existing silhouette determination algorithms that can be roughly classified as follows:

Image-Space Silhouette Algorithms

Image-space silhouette algorithms extract silhouettes by way of image-processing operators that detect discontinuities in specific image buffers, called G-Buffers [106] (Sec. 4.1). These buffers store the geometric properties of 3D geometries and result from conventional z-buffer rendering. Finally, silhouettes are represented in an image by pixels. Since the edge-based rendering algorithms introduced in this thesis are based on the G-Buffer concept, Chapter 4 gives a more detailed treatment.

The advantage of image-space algorithms is that they can be fully accelerated by today's graphics hardware. For example, Mitchell et al. present a technique that extracts silhouettes for enhancing images on a per-scene basis. They render silhouettes and outline regions in shadow and texture boundaries with their method (see Figure 2-3) [84].

Since image-space algorithms represent computed silhouettes merely by pixels, silhouettes lack an analytic representation. Hence, artistic stylization is usually difficult to achieve. However, Curtis generates sketches based on image-space silhouettes using a *loose and sketchy* filter [17]. The filter employs a *depth map*, that is, a 2D image that stores depth values at each pixel, as input and converts it into a *template image* and into a *force field* image. The template image determines the amount of ink needed in the neighborhood of a pixel. To generate sketches of various styles, particles are placed randomly in image space and move along the silhouettes, adding or erasing ink until they "die". The force field image thereby affects the movement of particles along edges. The results produced by Curtis' loose-and-sketchy filter can be used for superimposing images. For example, a layered composition of the scene's sketch, a blurry color image of the scene, and a procedural "paper" texture as background can simulate a hand-drawn



[Sousa et al. 2003] **[DeCarlo et al. 2004]** **[Teece 2003]**

Figure 2-2: **Computer-generated Line Drawings.** Well-placed lines often suffice to encourage viewers to complete a picture using their imagination.

style sketch on a rough paper (see Figure 2-3). Nevertheless, the loose-and-sketchy filter cannot be combined with arbitrary scene contents using z-buffering and is not meant to run in real-time.

Hybrid Silhouette Algorithms

Hybrid silhouette algorithms first apply operations in object space (e.g., to modify the model's polygons) and then render the output using operations in image space that affect the composition in the frame buffer (e.g., by marking regions in the stencil buffer for later use). Hybrid algorithms usually require several rendering passes for rendering silhouettes.

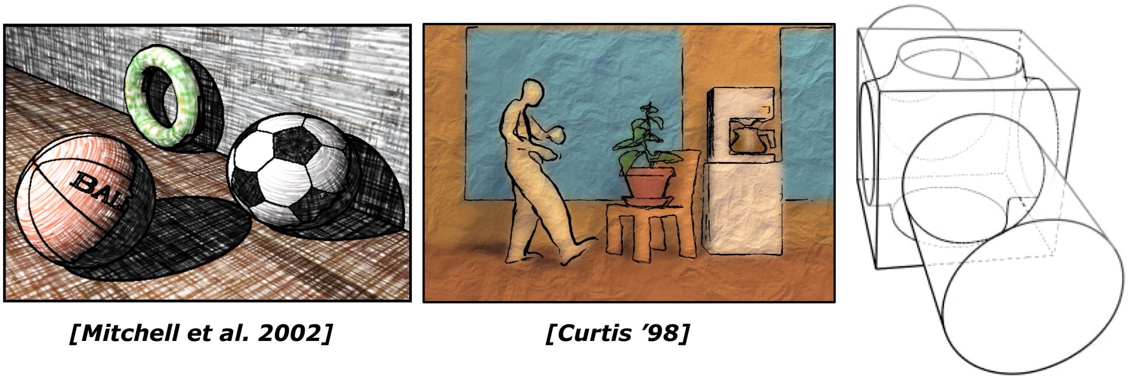
Rossignac and Emmerik employ the z-buffer to determine silhouettes [103]. In the first pass, they render the 3D model's polygons in a solid color (e.g., white). In the second pass they render the model's polygonal edges in black using a thick wire-frames style. For this, they shift polygons slightly in the z-direction for depth testing, yielding black silhouettes in image space. Raskar and Cohen present a generalization of the algorithm that also overcomes the problem of the z-buffer's numerical imprecision [101]. They first render all polygons using (enabled) back-face culling to fill the contents of the z-buffer with the surface's depth values. They then render all polygons in black using (enabled) front-face culling and the depth test set to "equal". In this way, a pixel wide line gets rendered wherever front-facing and back-facing polygons meet. To increase the width of silhouettes they slightly translate polygons towards the camera and set the depth test to "less-or-equal" when rendering back-facing polygons. The further enlargement of back-facing polygons with respect to the camera position results in a constant silhouette width. Raskar extends this approach by inserting additional geometric primitives, such as quads, instead of enlarging back-facing polygons [102]. Furthermore, he uses graphics hardware to adjust the size and orientation of the quads.

Hybrid algorithms are similar to image-space algorithms in that they represent silhouettes by pixels in image space. Hence, further artistic stylization is hard to achieve.

Object-Space Silhouette Algorithms

Object-space silhouette algorithms compute silhouettes entirely in object space. Because object-space algorithms determine silhouettes analytically they require additional algorithms that resolve the silhouettes' visibility.

A naive approach to silhouette detection processes each polygonal edge of a mesh. In this approach, edges that are adjacent to a polygon facing towards the camera and to a polygon



[Mitchell et al. 2002] **[Curtis '98]** **[Markosian et al.'97]**

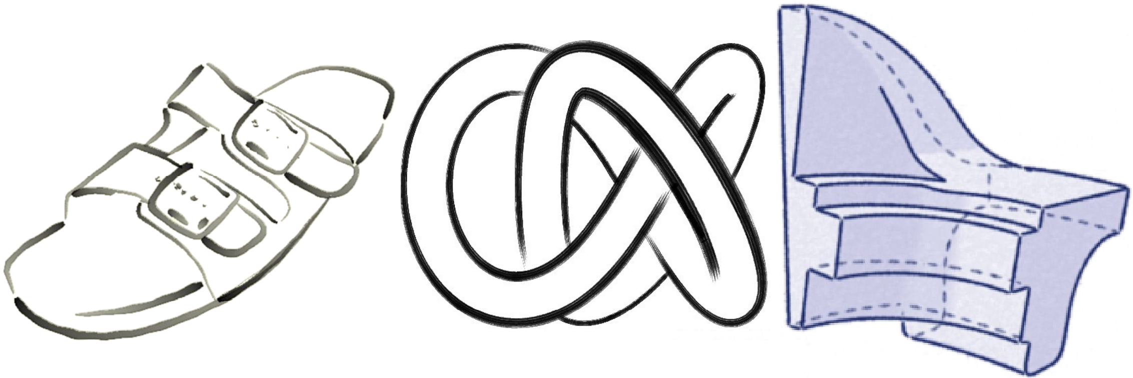
Figure 2-3: **Outlining and Sketching.** Rendering the outlines of 3D models demonstrates their distinctive display, can attract visually, and can communicate shapes.

facing away from the camera are considered to be valid silhouettes. BUCHANAN AND SOUSA present a data structure, the *edge buffer*, to reduce the computational complexity of silhouette detection [11]. The edge buffer stores two bits per edge to indicate the edge's adjacent polygons as front facing or back facing. The process of first iterating the set of polygons to process their orientation, and then updating the associated bits (using the XOR-operator) marks the edges as silhouettes for subsequent rendering. However, the computational cost for silhouette detection is either linearly dependent on the number of polygonal edges or linearly dependent on the number of polygons. This is generally too expensive for rendering high-tessellated 3D geometries in real time.

Alternatively, preprocessing methods for 3D geometries that set up efficient data structures exist for reducing the computational costs for silhouette detection at run time. BENICHOU AND ELBER [9] and GOOCH ET AL. [45] project the normal vectors of polygons onto a *Gauss map*, that is, a bounding sphere centered at the model's origin. They represent each polygonal edge by the arc on the sphere that connects both normal vectors' projections of the edge's adjacent polygons. In the case of an orthographic camera model, a plane perpendicular to the viewing direction passing through the origin intersects the arcs of those edges that represent silhouettes. To determine intersecting arcs efficiently at run time they set up a data structure that hierarchically decomposes the sphere. However, preprocessing is not suitable for animated polygonal meshes.

Based on the observation that virtual $O(\sqrt{n})$ edges of n polygons are silhouettes [105], MARKOSIAN ET AL. present a probabilistic algorithm that randomly selects a small fraction of polygonal edges and exploits their spatial coherence [77]. That is, whenever the algorithm detects a silhouette it recursively traces adjoining edges until it reaches the end of the entire chain of silhouettes. In this way, MARKOSIAN ET AL. can determine most silhouettes, but they cannot guarantee to find all of them (see Figure 2-3).

Silhouette algorithms in the previous categories compute the visibility of silhouettes directly by just employing depth testing with the z-buffer contents. This approach is not very helpful in the case of objects-space silhouettes, because part of them is likely to be clipped by the model's surface, in particular if the silhouettes have been further stylized. Object-space algorithms that analytically determine the visibility of silhouettes usually deploy APPEL's hidden line removal algorithm [7], which is based on the *quantitative invisibility* (*QI*) of a point. The QI of a point denotes the number of front-facing polygons between the point on the 3D geometry's surface and the camera. APPEL uses ray tests to determine a point's QI. MARKOSIAN ET AL. minimize the number of ray tests by only determining the QI at the vertices along a chain of silhouettes. Having observed that the QI only changes at special kinds of vertices, the so-called *cusp*



[Northrup and Markosian 2000] ***[Isenberg et al. 2002]*** ***[Kalnins et al. 2003]***

Figure 2-4: **Artistically Stylized Silhouettes.** Rendering silhouettes in an artistic way can simulate hand-drawn line drawings. Object-space silhouette rendering algorithms allows one to render using different line styles.

vertices, they find they can propagate a QI along a chain of silhouettes and implement their probabilistic algorithm [77] efficiently.

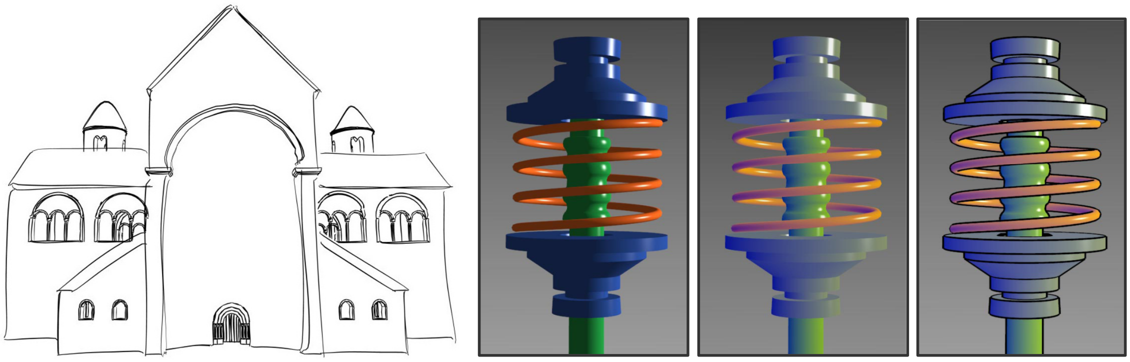
NORTHRUP AND MARKOSIAN implement an image-space approach for determining the visibility of silhouettes [87]. They render polygons and silhouettes by a unique color for generating an *id reference image* [78] (a.k.a. *id buffer*, Sec. 4.1) using conventional z-buffer rendering. They then iterate on the buffer's contents to identify those silhouettes that contribute to at least one pixel. Next, they scan-convert each visible silhouette to determine its visual portions, called *segments*. Using the segments' endpoints in image space, they generate an analytic representation of the silhouette path. ISENBERG ET AL. introduce a similar algorithm that manages visibility testing by sampling the z-buffer instead of the id buffer [58]. To reduce the numerical instability caused by the z-buffer's imprecision, they sample the neighborhood of a pixel's location as well. Furthermore, they consider the underlying mesh's connectivity information given by a winged-edge data structure to connect adjoining silhouette edges.

Object-space algorithms provide an analytic representation of silhouettes or of a chain of adjoining silhouettes. Once computed, silhouettes can be rendered by way of extra inserted geometry, such as triangle or quad strips. This allows one to stylize silhouettes using stroke-like rendered geometry that loosely aligns to the original 3D geometry in order to simulate artistic line drawings (see Figure 2-4). Furthermore, KALNINS ET AL. [59] introduce an algorithm that maintains temporal coherence for objects-space artistic silhouettes in interactive environments.

In conclusion, the analytic representation of silhouettes is essential for a user-defined, artistic stroke generation. This makes objects-space algorithms superior to both image-space and hybrid approaches. In addition, rendering visible as well as occluded objects-space silhouettes allows one to depict a 3D model's interior layout directly (see Figure 2-3 and Figure 2-4). In contrast, current image-space algorithms neither allow one to render occluded silhouettes nor to stylize silhouettes in interactive environments. However, image-space silhouette algorithms are independent of the number of a 3D model's polygons, whereas the computational cost of object-space algorithms reduces their efficiency, making them less practical for depicting 3D models of high geometric complexity.

2.2 Applications of Non-Photorealistic Rendering

The efficacy and expressiveness of non-photorealistic depictions depend particularly on the context they are used in, the intended connotation, and the information to be conveyed.



[Strothotte et al. '99] *[Gooch et al. '98]*

Figure 2-5: **Architectural and Technical Illustrations.** Sketches of medieval architecture can convey the uncertainty of knowledge. Technical illustrations communicate mechanical parts efficiently (simple Phong shading (left), cool-to-warm shading (middle), edge-enhanced cool-to-warm shading (right)).

Consequently, specific applications typically require meaningful non-photorealistic rendering techniques.

Architectural Visualizations

Architects produce architectural drafts by hand in order to present their design decisions, to document planning states, and to convey work-in-progress to their clients.

Schumann et al. present a first empirical study about the efficacy of non-photorealistic rendering in architectural design [107]. They approve the following hypotheses by interviewing architects:

1. For presentation of early drafts of architecture designs, sketches are preferred to CAD plots (i.e., exact line renderings) and shaded images (i.e., realistic renderings).
2. Sketches perform better in the communication of affective and motivational aspects, while exact plots and shaded images perform better for cognitive aspects.
3. Sketches stimulate viewers more than shaded images to discuss and actively participate in design development.

As a result, artists generally prefer to present sketches as first draft to the client because sketches illustrate the preliminary state of a design more effectively than realistic renderings. In contrast, exact line renderings and realistic depictions are preferred when persuading clients to accept a final design. With respect to hypothesis two, sketches are considered "more interesting, lively, imaginative, creative, and individual" and initiate "discussions and active changes" whereas exact line renderings are "more comprehensible, more recognizable, and clearer". Finally, both sketches and realistic renderings seem to encourage participants equally actively to discuss extensions and developments by way of "verbal descriptions", "gesturing or pointing to the image", or "drawing on another sheet of paper". However, sketches encourage more "drawing directly onto the image".

The study shows the following advantages of NPR: the artist can apply non-photorealistic rendering to avoid giving clients a false impression of completeness, to emphasize only certain aspects and, therefore, avoid raising discussions on irrelevant details, and to sketch multiple variants that help clients to choose the most appropriate one [116].

Furthermore, Strothotte et al. [114] argue for visualizing uncertainty explicitly by way of non-photorealistic rendering techniques (e.g., by sketching line drawings) to avoid misinterpreting depicted data. For example, in the case of a virtual reconstruction of ancient or medieval architecture, a photorealistic rendition might give a false impression of authenticity. That is to say, although there is no evidence in the image, the viewer of a realistic depiction might possibly assume that the image reveals proven and certain knowledge. Thus, encoding additional, non-geometric information, such as uncertainty, using non-photorealistic rendering styles can improve the reliability of visualizations. Figure 2-5 illustrates a virtual reconstruction in a sketchy style for communicating a possible variant of the building's layout.

Technical Illustrations

Instruction manuals, illustrated textbooks, and encyclopedia show illustrations of industrial and mechanical parts as well as technical information. Artists are sophisticated at producing illustrations that communicate efficiently.

GOOCH ET AL. introduce a set of non-photorealistic rendering techniques that use the principles of traditional art, such as line character, shading, and shadowing, to generate attractive and informative colored, technical illustrations automatically [45]. Amongst other things, they provide techniques that address the following two important visual cues for communicating shape: distinctive rendering of outer and interior edges, and surface shading ranging far from black and white in a cool-to-warm undertone.

Edges, such as silhouettes, object boundaries, and creases[1], are typically enhanced in technical illustrations to outline individual parts and to suggest important gestalts. Illustrators usually draw edges in black but often switch over to white to accentuate interior edges, such as creases, whenever using color shading to illustrate objects. Shading is typically used for communicating shape information (e.g., material and curvature) and its orientation. GOOCH ET AL. introduce a low dynamic-range lighting technique for technical illustrations [43] that lets viewers perceive shape information even in dark areas and at white highlights. Their technique ensures that black and white edges remain visually distinct from the shape's color. GOOCH ET AL. observed that in colored illustrations *tones*, that is, colors resulting from scaling with gray, vary much in hue but only a little in luminance. Their technique basically adds not only a subtle luminance variation but also a prominent temperature shift to the object's color. Thus, the varied color ranges from a value with a warm undertone, e.g., yellowish undertone, in lit regions to a value with a cool undertone, e.g., bluish undertone, in unlit regions (see Figure 2-5).

Smart Depiction Systems

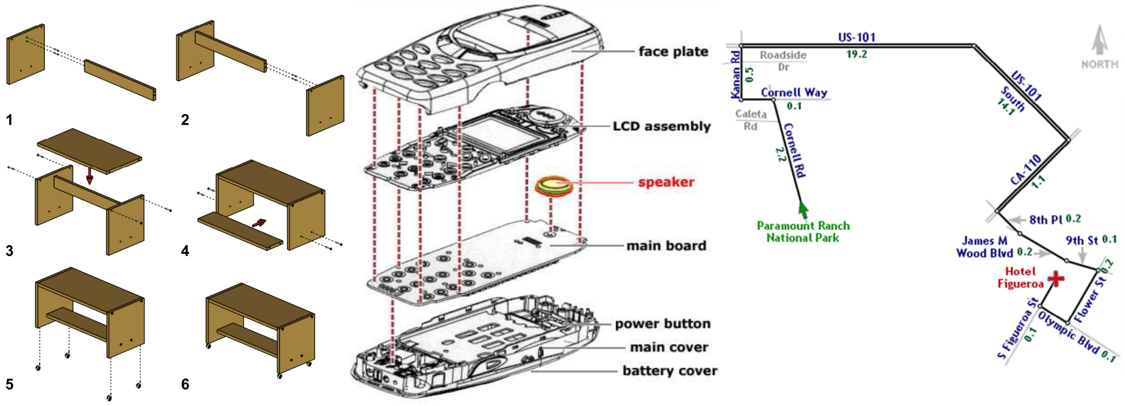
Smart depiction systems are computer algorithms and computer interfaces that use principles and techniques adopted from graphics design, visual art, perceptual psychology, and cognitive science to automatically generate visual displays for communicating information efficiently. In many domain-specific areas, smart depictions systems have been investigated that aim at reducing the time and effort required to generate rich and effective visual contents significantly [1]. Non-photorealistic rendering represents an enabling technology for implementing such systems that produce sophisticated depictions and diagrams.

In the domain of assembly instructions, AGRAWALA ET AL. [2] present a system that plans assembly operations and produces compelling step-by-step illustrations for assembling everyday objects. The algorithmic techniques are based on design principles derived from cognitive psychology research. In addition, they follow the conventions of technical illustrations for rendering the final assembly diagrams, e.g., using distinctive outlines for single parts (see Figure 2-6).

LI ET AL. implement a semi-automatic authoring tool to create interactive exploded view diagrams derived from 2D images [73] (see Figure 2-6). Their system enables one to annotate

---

[1] These edges are classified as a 3D model's visually important edges in Chapter 4.



[Agrawala et al. 2003] *[Li et al. 2004]* *[Agrawala and Stolte 2001]*

Figure 2-6: **Smart Depiction Systems.** Smart depiction systems automatically generate effective pictures and diagrams for assisting people: systems for generating assembly instructions, exploded view diagrams of assemblies, or route maps can reduce time and cost.

individual parts and to specify how parts of the assembly expand and collapse. The accompanying presenter allows one to browse the exploded view diagram, e.g., when searching for and identifying single parts.

Route maps are one of the most common graphic interfaces that depict paths and directions. AGRAWALA AND STOLTE introduce a system that designs and renders route maps in a hand drawn style automatically [3][4] (see Figure 2-6). They present cartographic generalization techniques based on the principles of mapmaking and the abstraction techniques found in hand drawn route maps.

Chapter 3

Hardware-Accelerated Computer Graphics

During the last few years, an important trend in computer graphics hardware design has been the increase in programmability and performance of graphics accelerators (a.k.a. *Graphics Processing Units* (*GPUs*)) [37]. Today's graphics hardware features a partially programmable rendering pipeline and offers efficient parallel computing power. In addition, software interface for graphics hardware, such as OpenGL, offer high-level shading languages, which utilize the GPU's flexibility, making real-time computer graphics a rapidly evolving field. This advance inspires researchers and graphics engineers to invent novel real-time rendering techniques: ideas-turned rather than capability-bound rendering algorithms are being developed. In a sense, graphics hardware evolution is influencing, or even reshaping image-synthesis algorithms and, in consequence, real-time computer graphics. So, the exploitation of today's graphics hardware capabilities is a prerequisite for real-time rendering algorithms in both the field of photorealism and the field of non-photorealism.

This chapter presents GPU programming concepts as well as the advanced real-time rendering techniques established in the last few years. These are essential for implementing the edge-enhancement algorithm, the blueprint rendering technique, and the sketchy drawings rendering technique presented in the remainder of this thesis. Section 3.1 outlines the "programmable rendering pipeline" and Section 3.2 outlines the OpenGL Shading Language. The chapter then proceeds with an overview of rendering techniques: Section 3.3 introduces the render-to-texture concept, Section 3.4 introduces image processing on graphics hardware, Section 3.5 introduces dependent-texture access, Section 3.6 introduces deferred shading, and Section 3.7 introduces depth-sprite rendering.

3.1 Programmable Rendering Pipeline

The *graphics rendering pipeline* represents a model for the synthesis process of 2D images based on general 3D scene descriptions. The rendering pipeline is conceptually divided into the following stages: (1) the *application stage* concerned with modeling the 3D scene and setting up the rendering process, i.e., by initializing the synthesis process and submitting 3D geometries as geometric primitives (such as polygons, triangles, or quads) to the next stage, (2) the *geometry stage* concerned with operating on a per-primitive and per-vertex basis, and (3) the *rasterization stage* concerned with breaking down primitives to fragments, assigning colors to them, and forwarding them to the frame buffer, so that they possibly constitute pixels in the final image. The application stage is generally managed on the CPU, whereas the actual image-synthesis process is typically implemented on graphics hardware to achieve real-time frame rates, that is, generating at least 6 images per second, or, *frames per second* (*fps*) [5].

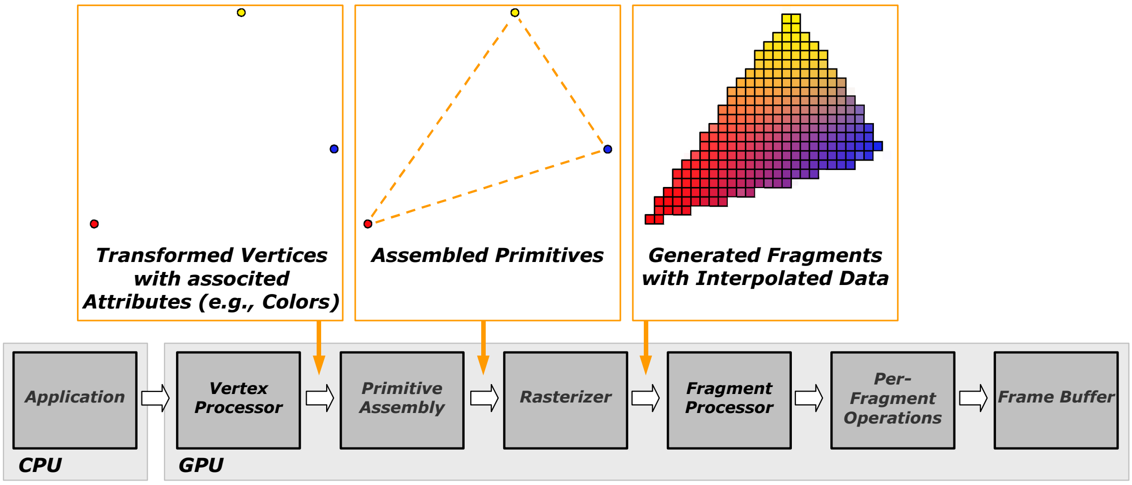


Figure 3-1: **Programmable Rendering Pipeline.** Vertex and fragment shaders replace the vertex processing stage and the fragment processing stage.

The term *programmable rendering pipeline* is somewhat misleading because only part of the rendering pipeline on graphics hardware is essentially programmable today. The term denotes that both the vertex processing stage and the fragment processing stage of the rendering pipeline are programmable. That is, computer programs, a.k.a. *shaders*, can be written for each stage, facilitating a user-defined processing of either vertices or fragments. However, shaders offer great flexibility for implementing rendering algorithms on graphics hardware.

Figure 3-1 provides an overview of the programmable rendering pipeline and the instances it processes. The pipeline is a sequence of stages that operate in a fixed order. Each stage receives its input from the preceding stage and then directs its output to the subsequent stage. Since the operations performed on each input datum of a stage remain identical throughout a single rendering pass and access across individual instances is not possible, both vertex and fragment processing stages are capable of operating on their data in parallel. Hence, a hardware implementation can, for instance, have multiple processors that operate in parallel. For example, the GeForce 6800 GT has 16 pixel-pipelines for processing fragments.

Vertex Shading

When using OpenGL, an application generally submits geometric primitives, such as triangles, polygons, or lines, to the GPU by defining their vertices explicitly together with their associated attributes within a `glBegin/glEnd`-statement [111].

A *vertex shader* is a GPU-program that operates on incoming vertices and their associated *vertex attributes*, such as normal vectors, color values, texture coordinates, or any kind of user-defined attributes such as binormal vectors or tangent vectors. The vertex shader is primarily responsible for transforming the vertex's position to clipping coordinates. Moreover, a vertex shader performs any optional operation to manipulate vertex attributes or to generate additional vertex attributes. A vertex shader typically determines the vertex's color value based on a given lighting model as well as its texture coordinates. In the end, a vertex shader must at least output the vertex's clipping space position. Optionally, it can output additional vertex attributes for the next stage.

When *primitive assembling* has been performed, that is, assembling vertices into geometric primitives using the primitives' connectivity information, the *rasterizer* splits the geometric primitives of continuous space into a set of fragments of discrete window space based on the

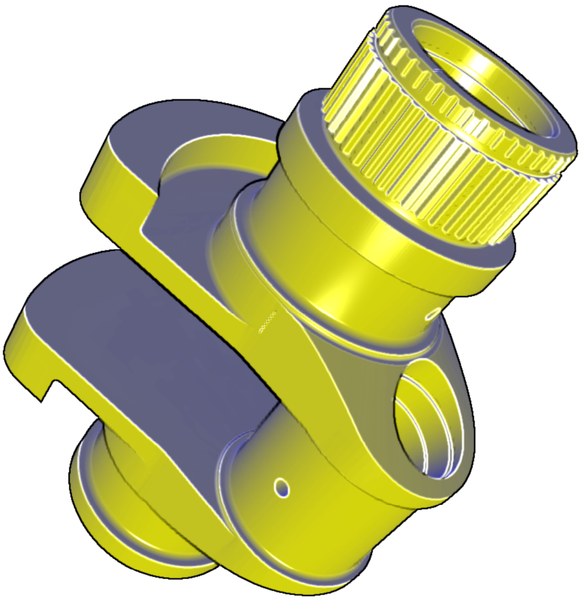


Figure 3-2: **Gooch Lighting Model for Technical Illustrations.** White edges outline inner forms while black edges outline the profiles of the crank. The image has been rendered with the edge-enhancement algorithm (see Chapter 4).

frame buffer's resolution. The rasterizer thereby interpolates vertex attributes using barycentric coordinates [38] and provides these attributes along with the associated fragments.

Fragment Shading

A *fragment shader* is a GPU-program that operates on each fragment produced by the rasterizer. A fragment provides its own raster position, its projective depth, and associated interpolated vertex attributes as input. Operations on fragments include, for instance, lookups into textures by way of texture coordinates for computing the fragment's final R, G, B, A color. A fragment shader then directs the fragment's final color value to the next state and can optionally submit the fragment's depth value too.

In contrast to a vertex shader, a fragment shader can discard fragments from further processing. If a fragment passes the shader regularly, its output values, that is, the fragment's final color and depth value, proceed to the *per-fragment operations*. These include, for instance, the alpha test, the stencil test, the depth test, and the blending operations. Finally, the fragment possibly emits the pixel at the fragment's raster position in the frame buffer. *Occlusion queries* can determine the number of fragments that effectively end up in the frame buffer by counting those fragments passing both the stencil test and the depth test [61][108].

As common functionality, both vertex and fragment shaders can request the current rendering context to retrieve information about the current OpenGL state, such as the modelview or projection matrices or active light sources. They can also access constant user-defined data provided along with the shaders as input. Furthermore, a set of mathematical and logical operations and functions are provided for computation.

Since vertex and fragment shaders replace the fixed functionality of the rendering pipeline stages, the computer graphics researcher or developer has to implement the elementary operations, such as vertex transformations, lighting calculations, or texture coordinate generations. Nevertheless, vertex and fragment shaders enable one to implement user-defined effects.

Until the advent of high-level shading languages such as *Cg* [37][76], and the OpenGL Shading Language [104], which started in 2003, vertex and fragment shaders were exclusively programmed using an assembly language [61].

3.2 The OpenGL Shading Language

The *OpenGL Shading Language* [104] is a C-like high-level shading language specifically designed for the OpenGL Architecture [108][111]. Its functionality has been inspired by the *RenderMan Shading Language* [6][121]. Unlike the RenderMan Shading Language, which lets one implement various shaders for surface shading, light sources, or atmosphere effects, the OpenGL Shading Language merely allows one to implement a single shader program that is in charge of considering all the interacting effects at once. Shader programs are then made part of the current rendering state of the rendering context of OpenGL [104]. Consequently, only one program can be active at any one time.

Shader programs consist of shaders that implement either a vertex or a fragment shader (Sec. 3.1). A typical OpenGL shading program contains one of both kinds each of which defines the function `main()` as the entry point into the shader. Listing 3-1 illustrates the vertex shader and Listing 3-2 illustrates the fragment shader for implementing the Gooch lighting model (see Chapter 2, [43]) using the Phong shading model [5] (see Figure 3-2). The remainder of this section gives an overview of the OpenGL Shading Language needed for upcoming code snippets.

The language defines built-in data types, such as `float`, `vec3`, or `mat3`, to support scalars, vectors, and matrices and handler types, such as `sampler2D`, that usually indicate textures.

It provides built-in functions, e.g., geometric functions; such as `normalize()` for normalizing vectors, or texture access functions, such as `texture2D()` for looking up into a `sampler`, or texture. For convenience, the function `ftransform()` (see Listing 3-1) returns the clipping coordinates of a vertex for output.

Listing 3-1: **Vertex Shader for Implementing the Gooch Lighting Model.** The shader directs eye-space vertex positions, normal vectors, and surface color values to the rasterizer for the fragment as input.

```
varying vec3 eyePos;          // Eye-space position of a vertex
varying vec3 eyeNormal;       // Eye-space normal vector of a vertex

void main(void) {
    eyeNormal     = normalize(gl_NormalMatrix * gl_Normal);
    eyePos        = normalize(vec3(gl_ModelViewMatrix * gl_Vertex));
    gl_FrontColor = gl_Color;
    gl_Position   = ftransform();
}
```

The shading language provides the following qualifiers for declaring variables that can communicate with shaders:

The `attribute` qualifier denotes vertex attributes provided together with a vertex when specifying geometric primitives. They thus represent input data for a vertex shader. Special built-in input variables exist for accessing the vertex's attributes in a vertex shader (e.g., `gl_Vertex` for accessing the vertex's position, `gl_Normal` or `gl_Color` for accessing its normal vector and color, which are usually specified by `glNormal` or `glColor` within the `glBegin/glEnd`-statement).

The `uniform` qualifier denotes user-defined variables that serve as input data for vertex or fragment shaders. These variables remain constant during shader execution, that is, they can only change per primitive. The material properties of 3D geometry, such as the warm color for the Gooch lighting model, represent commonly used `uniform` variables. Built-in `uniform` variables let shaders access the settings of the OpenGL state (e.g., `gl_ModelViewMatrix` and `gl_TextureMatrix` provide the current modelview matrix or texture matrices and `gl_LightSource` provide information about the light source.)

The `varying` qualifier denotes variables that communicate results form a vertex shader to a fragment shader. That is, these variables form the output of a vertex shader, then get interpolated by the rasterizer, and finally serve as input for the fragment shader. Built-in `varying` variables exist for directing vertex attributes to fragment shaders, (e.g., `gl_FrontColor` submits a vertex's front-facing color and `gl_TexCoord` its texture coordinates). A vertex shader can write to special output variables; it must write at least to `gl_Position` to submit the vertex's clipping coordinates. A fragment shader uses built-in variables to access interpolated `varying` variables (e.g., `gl_Color` or `gl_TexCoord` for accessing the fragment's color or texture coordinate). In addition, special input variables exist (e.g., `gl_FragCoord` provides a fragment's window relative *x* and *y* coordinates and perspective depth). A fragment shader merely supports the following two (special) output variables used for submitting fragments to the next stage: a well-defined fragment shader must write to `gl_FragColor` for passing a fragment's color, whereas, it can write to `gl_FragDepth` to replace the a fragment's projective depth value for depth testing. In addition, the keyword `discard` allows the shader to reject fragments during fragment shader execution.

Listing 3-2: **Fragment Shader for Implementing the Gooch Lighting Model.** The shader enhances inner edges and profile edges differently by considering edge intensities provided as 2D texture, the edge map (see Chapter 4).

```
varying vec3 eyePos;         // Eye-space position of a vertex
varying vec3 eyeNormal;      // Eye-space normal vector of a vertex
uniform vec3 warmColor;
uniform vec3 coolColor;
uniform float diffuseWarm;
uniform float diffuseCool;
uniform sampler2D edgeMap;   // 2D texture with edge intensities
uniform vec2 windowDimension;// Canvas resolution

void main (void) {
    // Sampling the edge map to access edge intensities
    vec4 tex = vec4(gl_FragCoord.x / windowDimension.x,
                    gl_FragCoord.y / windowDimension.y, 1.0, 1.0);
    tex = gl_TextureMatrix[0] * tex;
    vec3 edgeIntensity = texture2D(edgeMap, tex.xy).xyz;

    // Use the position of the first light source for Gooch Lighting
    vec3 L      = normalize(vec3(gl_LightSource[0].position.xyz));
    vec3 V      = normalize(eyePos);
    vec3 N      = normalize(eyeNormal);
    vec3 R      = normalize(reflect(L,N));
    float NdotL = (dot(N,L) + 1.0) * 0.5;

    // Calculate cool and warm colors
    vec3 kcool  = min(coolColor + diffuseCool*vec3(gl_Color), 1.0);
    vec3 kwarm  = min(warmColor + diffuseWarm*vec3(gl_Color), 1.0);
    vec3 kfinal = mix(kcool, kwarm, NdotL);
    float spec  = max(dot(R,V), 0.0);
    spec        = pow(spec, 32.0);
    vec4 goochColor = vec4(min(kfinal+spec, 1.0), 1.0);

    // Enhance inner edges and profile edges differently
    vec4 white = vec4(1.0,1.0,1.0,1.0); // Whitened inner edges
    goochColor = mix(white, goochColor, edgeIntensity.x);
    goochColor = goochColor * edgeIntensity.y; // Darkend profiles
    gl_FragColor = goochColor;
}
```

Once vertex and fragment shaders have been written in the OpenGL Shading Language, they can be linked to a single shader program and compiled for the targeted graphics hardware. That is, the assembly code for both vertex and fragment processing stages will be generated and can then be transferred to the GPU.

The real-time rendering techniques presented in this work operate mostly on fragments. In particular, blueprint rendering and sketchy drawing are based on a user-defined processing of fragments.

3.3 Render-to-Texture

When synthesising the image of a 3D scene the result is typically rendered into the frame buffer that will be displayed. Advanced rendering algorithms frequently require the rendering results of a preceding rendering passes as textures to implement complex effects (e.g., shadow mapping requires the construction of a depth texture [109]). For this, a *copy-to-texture* can be used, that is, the scene gets rendered into a non-visible frame buffer, such as a *pbuffer* [61], and copied to a 2D texture. Although a copy-to-texture is efficient because the frame-buffer contents remain on the GPU, a render-to-texture operation accelerates the creation of texture contents on current graphics hardware. For *render-to-texture*, a texture serves as a render target so that the GPU can write directly to it. A *render target* is loosely defined as the destination for synthesising the image of the 3D scene. Both a pbuffer and a 2D texture represent valid render targets. Furthermore, *multiple render targets* allow one to render the same 3D scene to several render targets simultaneously using different fragment color values for synthesizing individual images. For this, all render targets must be of identical dimensions but can be of different formats. For example, one can think of a render target storing three 8-bit scalars for *R*, *G*, *B* color values (e.g., for providing material properties as color values) and an additional render target storing 32-bit floating-point scalar values (e.g., for providing high-precision linearized depth values as texture). Textures that can have up to 32-bit floating-point precision per color channel are referred to as *high-precision textures* (a.k.a. *float textures*) [61]. In conclusion, multiple textures storing different information about the 3D scene can be computed simultaneously.

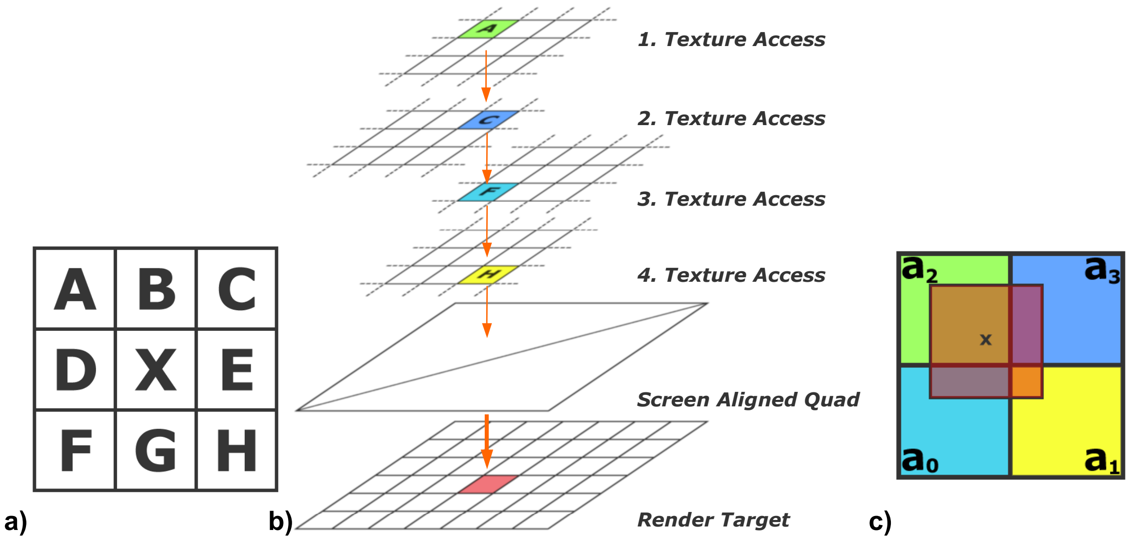


Figure 3-3: **Sampling Neighboring Texture Values.** (a) A 3×3-filter kernel defines an area around a pixel covering its neighboring pixels. (b) Shifting texture coordinates of a fragment diagonally allows one to sample neighboring texels for implementing image processing on graphics hardware. (c) A texture lookup using linear filtering returns a weighted average of four neighboring texels' values.

Render-to-texture and multiple render targets represent fundamental concepts for implementing a variety of real-time rendering techniques today. In particular, deferred shading (Sec. 3.6) becomes feasible in real-time.

In the emerging research area *General-Purpose Computations Using Graphics Hardware* (a.k.a. *GPGPU*) the GPU gets exploited not for the image-synthesis process but for general-purpose computations [46][51]. The techniques introduced in this field employ the render-to-texture capabilities and multiple render targets to store the computed data as textures, and to reuse them as input for subsequent calculations. GPGPU techniques exist, for instance, for implementing physically based simulations [50] or for numerical calculations [68].

The rendering techniques introduced in this thesis employ the render-to-texture concept and multiple render targets to capture rendering results efficiently for subsequent use.

3.4 Image Processing on Graphics Hardware

Image-processing operations process the pixels of a 2D image, the *source image,* to implement an image operation resulting in a *destination image*. A single source pixel can determine the color of the destination pixel (*point operation*), or multiple (often neighboring) pixels of the source image can determine the color of the destination pixel (*local operation*) [42]. Performing image processing using the CPU is typically time consuming and hardly possible in real-time.

Image processing on graphics hardware represents the mapping of 2D image operations to the process of synthesizing images of 3D scenes, and seamlessly integrates into the programmable rendering pipeline. As a result, image processing can now operate in real-time [82][83].

2D textures that either represent 2D images or result from first rendering 3D scenes to a render target (Sec. 3.3) form the basis for performing image-processing operations. Mapping a texture to a screen-aligned quad, that is sized so that the image will render onto the screen mapping texels to pixels, facilitates image processing. A fragment shader that then operates on each fragment resulting from rasterizing the quad, samples the texture to implement image operations. The fragment shader calculates a fragment’s texture coordinates and then samples the texture, possibly multiple times, to compute the fragment’s final color. The results are then directed to the render target that finally stores the processed image.

In the case of a point operation, a fragment shader uses the texture coordinates determined by the fragment’s screen-space position multiplied by the reciprocal of the texture size to access the associated texel. In this way, *nearest filtering* ensures a virtual 1-to-1 mapping between pixels and texels. That is, the value of the texel nearest to the sample position, or, to the texture coordinates serves as the texture lookup’s result. The fragment shader then operates on the resulting color value to implement, for instance, contrast enhancements, color space transformations [122], or sepia tones.

Neighbor sampling

Local operations require the values of a texel’s neighboring texels for computation. A fragment shader thus needs to sample one and the same texture multiple times. The texture coordinates that correspond to the adjacent texels’ locations result from shifting the previous texture coordinates by a multiple of a texel’s size in the x and y directions in screen space.

Filter kernels usually define the region around a pixel’s location. The surrounding pixels are considered for computation. In addition, filter kernels define weights to specify the contribution of each pixel value to the entire result. For example, a 3×3 filter for blurring an image’s contents uses a ninth part of the original pixel’s value and the values of its eight adjacent pixels. The grid in Figure 3-3a illustrates the layout of a 3×3 filter kernel where pixels A-H represent neighboring pixels adjacent to pixel X.

Shifting texture coordinates diagonally and then sampling one and the same texture four times allows one to access the texture values of texels A, C, F, and H that are adjacent to X (see Figure 3-3b). It should be noted that these four samples are sufficient to implement a 3×3 blur-filter when using bilinear filtering [5].

Listing 3-3: **Fragment Shader for Implementing Image Blur.** Sampling a texture at only four neighboring positions suffices to implement a 3×3 blur-filter.

```
uniform sampler2D image;  // Image texture
uniform vec2 dim;         // Image’s width and height

void main (void) {
    // Texture Center and Offset
    vec2 ctr = vec2(gl_FragCoord.x/dim.x,gl_FragCoord.y/dim.y);
    vec2 off = vec2((1.0/dim.x)*2/3,(1.0/dim.y)*2/3);

    // Access in direction A
    vec4 tex = gl_TextureMatrix[0]*vec4(ctr.x-off.x,ctr.y+off.y,1.0,1.0);
    vec4 A = texture2D(image, tex.xy);

    // Access in direction C
    tex = gl_TextureMatrix[0] * vec4(ctr.x+off.x, ctr.y+off.y, 1.0, 1.0);
    vec4 C = texture2D(image, tex.xy);
    // Access in direction F
    tex = gl_TextureMatrix[0] * vec4(ctr.x+off.x, ctr.y-off.y, 1.0, 1.0);
    vec4 F = texture2D(image, tex.xy);

    // Access in direction H
    tex = gl_TextureMatrix[0] * vec4(ctr.x+off.x, ctr.y-off.y, 1.0, 1.0);
    vec4 H = texture2D(image, tex.xy);

    // Output blurred destination image pixels
    gl_FragColor = vec4(0.25*(A+B+C+D));
}
```

Bilinear Texture Filtering

For each texture sample *bilinear filtering* finds the four closest neighboring texels around the sample position and then linearly interpolates their values to determine a weighted average. Assume (s,t) to be the texture coordinates for accessing a texture (e.g., the center of the red box marked by the cross in Figure 3-3c). Let (u,v) be the decimal parts of the texture coordinates computed by $u = s - \lfloor s \rfloor$; $v = t - \lfloor t \rfloor$ and let a_0, a_1, a_2, and a_3 be the neighboring texels' values. Bilinear filtering determines the interpolated average value a for the texture lookup as follows:

$$a = (1-u)\cdot(1-v)\cdot a_0 + u\cdot(1-v)\cdot a_1 + (1-u)\cdot v\cdot a_2 + u\cdot v\cdot a_3$$

Listing 3-3 illustrates the fragment shader for implementing image blur by sampling just four adjacent texture positions in A, C, F, and H direction. Here, shifting texture coordinates diagonally for sampling the texture provides interpolated texture values whose scaled sum constitutes the average of the values of the nine texels of the 3×3 grid.

Sampling neighboring texture positions for implementing image processing on graphics hardware enables image-space edge detection (see Chapter 4).

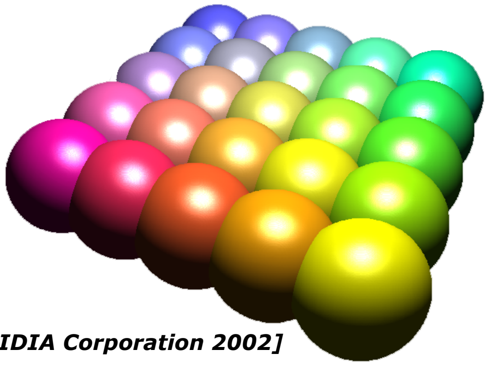
3.5 Dependent Texture Access

A *dependent texture access* (a.k.a. *dependent texture read*), denotes a texture access that uses texture coordinates that do not result from interpolating vertex attributes but from a user-defined calculation, e.g., within a fragment program [5]. For example, the result of a first texture access can form the basis for performing another texture access. A typical dependent texture access is to alter texture coordinates by way of an *offset texture*. That is, texture coordinates provided by the rasterizer first access the offset texture. The resulting texture value then offsets a second texture lookup into another texture. As one application, dependent texture accesses can be used to implement soft shadow algorithms [67].

In this thesis, a dependent texture access is basically used to perturb texture coordinates in image space to implement sketchy drawing (see Chapter 6).

3.6 Deferred Shading

In general, rendering techniques perform surface shading calculations directly on a per-fragment basis after rasterizing 3D geometry. However, complex shading effects typically require multiple rendering passes in order to accumulate the final colors in the frame buffer. For this, they render 3D geometry multiple times, which can be particularly time consuming. Furthermore, fragments of the 3D geometry's occluded faces get shaded as well. So, unneeded shading calculations are performed.



[NVIDIA Corporation 2002]

Figure 3-4: **Depth Sprites Substituting Lit Spheres.** A single depth sprite textured with geometric properties for lighting calculations represents each of the lit spheres.

Deferred shading [49] represents a rendering technique that performs shading calculations only to a 3D geometry's visible parts in a post-process. That is, shading calculations are applied to only the relevant pixels in the frame buffer after synthesizing the 3D geometry's image. For the technique to work, it requires the geometric properties (e.g., position, normal vector, light direction vector, and material parameter) at each pixel (e.g., to implement lighting calculations).

DEERING ET AL. introduce deferred shading for the *PixelFlow Architecture* [23] and PEERCY ET AL. uses deferred shading to implement *interactive multi-pass programmable shading* [98]. To implement deferred shading on today's graphics hardware, the geometric properties are first encoded on a per-fragment basis as color values. The resulting color values then populate 2D textures possibly using multiple render targets. Finally, the 2D textures serve as input for texturing a screen-aligned, screen-sized quad and a fragment shader can process the geometric properties provided for fragments of the quad [110]. In this way, complex shading calculations can be performed by rendering just a single quad (possibly multiple times).

Screen-aligned quads, for instance, represent lit spheres in Figure 3-4. Lighting calculations are performed based on the positions, normal vectors, and light direction vectors encoded in textures. In this thesis, preserving geometric properties in textures allows the edge-enhancement algorithm to extract a 3D model's edges in image space (see Chapter 4).

3.7 Depth Sprite Rendering

Depth sprites are, from a conceptual point of view, 2D images with an additional depth component at each pixel for depth testing. *Depth sprite rendering* fills the contents of the color buffer with the pixels' color values of the depth sprite whose associated depth value passes the depth test. In this case, the pixels' depth value will generally be written to the z-buffer as well. With respect to depth, depth sprite rendering behaves in a similar way to ordinary 3D scene rendering.

Based on high-precision textures (Sec. 3.3) and fragment shading, depth sprite rendering can be implemented as follows:

1. A high-precision texture containing depth values, called *depth map*, maps onto a screen-aligned quad, whereas a 1-to-1 relationship between pixels and texels is ensured.
2. A fragment shader then replaces the fragments' z-values of the quad (produced by the rasterizer) with the corresponding depth values derived from accessing the depth map. For optimizing fill rate, the fragment shader additionally discards fragments whose depth value equals 1, which denotes the depth of the back clipping plane. Otherwise, the fragment shader calculates the R, G, B, and A color values of the fragments.

3. Rendering then proceeds with the ordinary depth test. If fragments pass the test, the frame buffer will be populated with the color and depth values of the depth sprite.

It should be noted that depth sprite rendering requires nearest texture filtering for accessing the depth map to avoid low-pass filtering of depth values.

As one application, depth-sprite rendering allow one to resolve the visibility of image-based renderings, for instance, to implement impostors [5]. Furthermore, rendering 2D image data as depth sprites and employing deferred shading allow one to substitute 3D geometry while still providing dynamic lighting calculations (see Figure 3-4).

In this thesis, depth sprite rendering has been used to combine blueprints (see Chapter 5) and sketchy drawings (see Chapter 6) with arbitrary 3D scene contents.

Chapter 4

Real-Time Edge-Enhancement Rendering Algorithm

Outlining 3D models and their distinct features in depictions plays a major role for visual perception, it facilitates the communication of shape effectively. Non-photorealistic rendering algorithms often rely on edge-enhancement strategies, such as for silhouettes, border edges, and crease edges, for synthesizing comprehensible depictions, such as scientific, technical, or expressive illustrations (see Chapter 2). Thus, detecting the 3D models' visually important edges and enhancing them are fundamental tasks we have to address in real-time rendering.

For this reason, this work provides a rendering algorithm that encapsulates edge detection for edge enhancement as a generic component for real-time algorithms. Such a component simplifies the implementation of a wealth of rendering algorithms, such as non-photorealistic rendering algorithms. As a fundamental requirement, the edge detection for the edge enhancement component must be integrated into the 3D rendering pipeline and operate in real-time. Exploiting graphics hardware to implement an efficient, accelerated edge-enhancement algorithm is thus the primary goal.

Image-space algorithms for edge detection exploit discontinuities in the image buffers that result from ordinary 3D rendering. They extract edges by way of image-processing operators. Although image-space algorithms generally inhibit the stylization of edges and the detection of the 3D models' occluded edges, they offer the following advantages:

- They are stable and robust with respect to errors in the models' underlying polygonal meshes.
- They do not depend on the requirements of the meshes' topology.
- Their performance is independent of the models' geometric complexity (Sec. 2.1, [57]).

The use of image-space edges for image-synthesis results in homogenous and generalized visual depictions and facilitates the distinctive display of 3D models in 3D scenes. In addition, image processing using graphics hardware allows one to implement a real-time capable edge-detection technique.

This chapter introduces a real-time edge-enhancement rendering algorithm that extracts and preserves the 3D models' visually important edges and enhances these edges further [95]. The multipass-rendering algorithm utilizes graphics hardware to implement edge-detection operations in image space and to encode visually important edges as intensity values in an image texture. In this way, the algorithm can serve as a single rendering component, that is, it

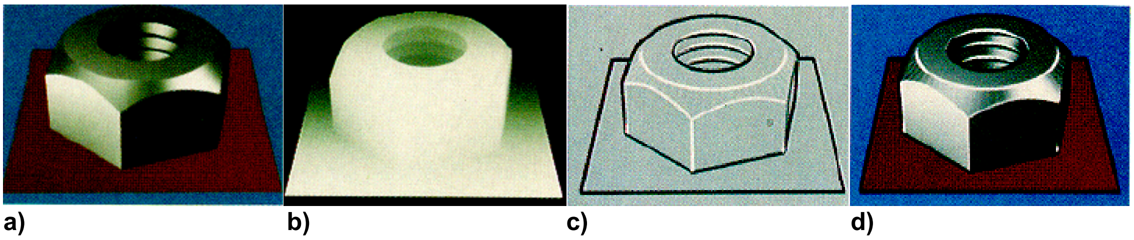


Figure 4-1: **Enhancing the Edges of 3D Models.** (a) The model of the nut rendered using traditional lighting and shading and (b) rendered using depth values. (c) 0th order discontinuities (black) and 1st order discontinuities (white) in the depth buffer show the profiles and internal edges. (d) Enhancing the nut's original shaded image produces a *Comprehensible Rendering of the 3-D Shape* [106]. (Images by SAITO AND TAKAHASHI [106])

seamlessly integrates both edge detection and edge enhancement into the programmable rendering pipeline in order to enable general use in real-time applications.

The remainder of this chapter is structured as follows: Section 4.1 describes the G-Buffer concept and related operators for enhancing the edges in images of 3D scenes. Section 4.2 classifies visually important edges of 3D models considered for subsequent enhancement. The algorithm itself is conceptually divided into two parts: Section 4.3 presents the part for detecting visually important edges and for preserving them as an image texture; Section 4.4 then illustrates how the texture can be reused for enhancing 3D models. Section 4.5 reviews the types and number of rendering passes that occur in the multipass-rendering algorithm. Section 4.6 presents applications illustrating the algorithm's applicability. Section 4.7 discusses optimization strategies and visual results and Section 4.8, finally, presents its application to the illustrative visualization of 3D city models. Illustrative 3D city models benefit directly from the edge-enhancement algorithm.

4.1 G-Buffer Concept

SAITO AND TAKAHASHI introduce *Geometric Buffers*, a.k.a. *G-Buffers*, as 2-dimensional data structures that store the geometric properties of 3D geometries [106]. Each G-Buffer of an extensible set of G-Buffers represents one category of geometric property such as normal vectors or depth values. In order to synthesize the G-Buffers' contents, the geometric properties of the 3D geometries' visible surface are rendered instead of surface colors, that is, geometric properties are directed to the pixels of the associated buffers. A G-Buffer can thus be considered as a 3D geometries' 2D image or as an intermediate rendering result. It can be seen that a single pixel of a G-Buffer represents one geometric property as a fraction of a 3D geometry's visible surface. Among many possible categories, frequently used G-Buffers are the following:

- The *depth buffer*[2] stores the depth values of visible surface fragments measured from the camera position.
- The *normal buffer* stores the normal vectors of visible surface fragments.
- The *id buffer* either stores the object identifiers that mark each single visible object of the 3D scene, or the surface patch identifiers that mark each single patch (e.g., a triangle or a polygon) of a surface's visible mesh.

Color images allow one to implement G-Buffers. For this, the *R*, *G*, *B*, *A* color channels store the scalar values of the geometric properties. Because a color image's color channel is typically

---

[2] We use the term depth buffer when referring to the concept of G-Buffers. If we refer to the visible surface determination algorithm we use the term z-buffer.

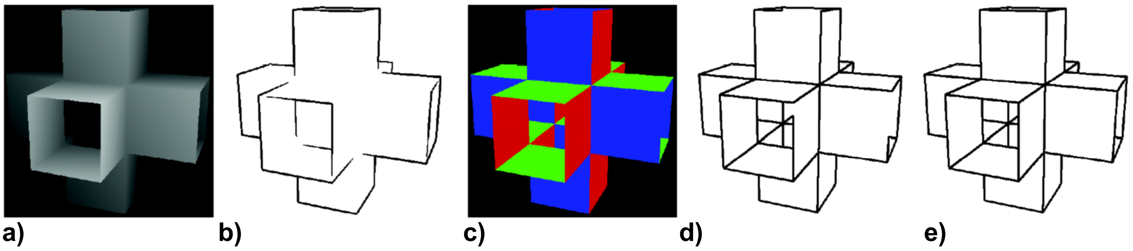


Figure 4-2: **Discontinuities in Depth and Normal Buffers.** The depth buffer (a) allows one to detect discontinuities (b) and the normal buffer (c) allows one to detect discontinuities (d). Combining the discontinuities of both G-Buffers results in a depiction of the model's edges (e). (Image by HERTZMANN [53])

limited with respect to precision and usually unsigned, geometric properties need to be encoded appropriately for later use.

The fundamental idea underlying the G-Buffer concept is their use as input for *enhancement operators* to derive artificial enhancements from 3D geometry. Operators exist, for instance, to extract *discontinuity edges*, *contour lines*, and *curved hatches*. Image-processing operators can implement these enhancement operators by processing each of the G-Buffers' pixels.

The discontinuity edges of 3D geometries, that is, the profiles and internal edges, outline the 3D geometries' outer boundary and enhance those edges that fall within the boundary as viewed from a certain viewpoint. Here, profiles correspond to 0^{th} order discontinuities and internal edges correspond to 1^{st} order discontinuities in the depth buffer. Applying 1^{st} and 2^{nd} order differential operators to the depth buffer allows one to detect them. As 1^{st} differential operator SAITO AND TAKAHASHI recommend the following linear filter, the Sobel Filter [97]:

$$I_0 = \frac{1}{8}\left(|A + 2B + C - F - 2G - H| + |C + 2E + H - A - 2D - F|\right)$$

and as 2^{nd} differential operator they suggest the following linear filter:

$$I_1 = \frac{1}{3}\left(8X - A - B - C - D - E - F - G - H\right)$$

Both operators process each pixel value (X) of the G-Buffer by further taking into account its neighboring pixel values (A–H) (see Figure 3-3a). In this way, they extract gradient values I_0 and I_1 for each pixel, forming discontinuity edges in the destination image. In order to correct "undesirable artifacts" in the destination image, e.g., large discontinuity changes and double line artifacts, SAITO AND TAKAHASHI implement a non-linear normalization operator that processes the minimum and maximum neighboring differential values [106].

Finally, the results in the destination image are combined with the 3D model's image to yield artistic enhancements. That is, superimposing the discontinuity edges on the model's original shaded image enrich it and generate "comprehensible" depictions of 3D geometries. Figure 4-1 illustrates the original shaded image, the depth buffer, the profiles and internal edges, and the resulting enhanced depiction.

DECAUDIN [21][22] extends SAITO AND TAKAHASHI's work to avoid visual inaccuracies, such as double line artifacts, produced by the 2^{nd} order differential operator applied to the depth buffer. He and later HERTZMANN [53] suggest detecting 0^{th} order discontinuities in the normal buffer instead of 1^{st} order discontinuities in the depth buffer. For this, they apply a 1^{st} order differential operator to the normal buffer to detect internal edges. In conclusion, processing the depth and the normal buffer enables one to detect both profiles and internal edges to enhance the image of a 3D model (see Figure 4-2).

4.2 Edge Classification

The edge-enhancement technique presented in this work enhances the visually important edges of 3D models in order to convey shape. For this, it considers silhouette edges, border edges, and crease edges as *visually important edges* of the 3D models.

Assuming that polygonal meshes constitute a 3D model's geometry without intersecting one another and without being clipped (see Figure 4-3a), their polygonal edges, that is, the edges connecting two vertices, can be classified as visually important edges as follows:

- A *silhouette edge* is an edge adjacent to one polygon facing towards the camera (*front facing*) and to one polygon facing away from the camera (*back facing*).
- A *border edge* is an edge adjacent to exactly one polygon.
- A *crease edge* is an edge between either two front facing polygons, or two back facing polygons whose dihedral angle is above some threshold.

Object-space algorithms exist for detecting visually important edges and determining their visibility (Sec. 2.1). Here, crease edges and border edges can even be detected in a pre-process as long as the mesh's triangulation remains unchanged. In contrast, silhouette must be computed for each frame when interacting with the scene, because they depend on the position and orientation of both the camera and the model. In general, the computational cost and complexity of object-space algorithms depends on the number of polygons or of the edges that connect vertices. Since hardware capabilities are still evolving, the geometric complexity of 3D models and 3D scenes is likely to increase as well. Thus, object-space edge-detection algorithms may become less appropriate for the real-time rendering of high-tessellated 3D models and 3D scenes of high geometric complexity.

3D models typically consist of multiple meshes that may possibly intersect one another. For example, the bangle in Figure 4-3b intersects the corpus of the Ogre. Furthermore, part of the surface may be clipped, e.g., by clipping planes or auxiliary geometry for the purposes of modeling (Sec. 5.6). In either case, additional polygonal edges are generally not added to the polygonal representations of the 3D models. So, edges that are visually important to perceive shape can occur without in any way corresponding to the meshes' polygonal edges. The previous classification is thus not sufficient to define visually important edges ambiguously, and it needs to be adjusted as follows:

- A junction where two polygons adjoin, whereas one polygon is visible and the other one is occluded along the junction (i.e., the visibility of the surface changes), represents a *silhouette edge*.
- A boundary of a polygon where no polygon adjoins represents a *border edge*.
- A junction where two polygons adjoin, whereas both polygons are visible along the junction (i.e., the visibility of the surface remains unchanged), and form a certain angle above some threshold represents a *crease edge*.

Figure 4-3a illustrates the visually important edges of a crank model consisting of multiple meshes linked to one another without any intersections. In contrast, Figure 4-3b illustrates the model of the ogre where meshes intersect one another producing visually important edges. These edges are not modeled explicitly and do not necessarily correspond to polygonal edges; most object-space edge-detection algorithms can hardly determine them efficiently without re-meshing the model's geometry.

Based on SAITO AND TAKAHASHI, a more descriptive classification considered by the edge-enhancement algorithm can be given: both silhouette and border edges represent *profile edges* that outline the (inner and outer) contours of 3D models; whereas crease edges represent *inner edges* that show inner features, such as changes of their surfaces' orientation. In a typical 3D

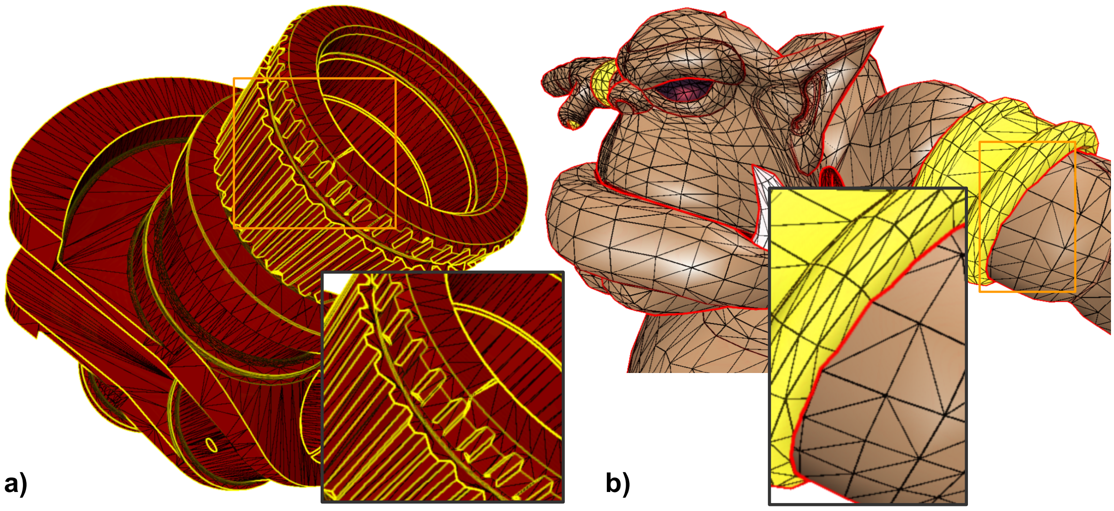


Figure 4-3: **Polygonal Edges and Visually Important Edges.** (a) The meshes that constitute the polygonal model of the crank do not intersect one another. Visually important edges (yellow) align to polygonal edges (black). (b) The mesh that sets up the bangle intersects the polygonal model of the ogre. Here, visually important edges (red) do not correspond to polygonal edges (black).

scene, abrupt changes in the depth buffer occur at silhouette edges and border edges, that is, 0^{th} order discontinuities of the depth buffer indicate profile edges. A 1^{st} order differential operator applied to the depth buffer detects them. Abrupt changes in the normal buffer typically occur at crease edges. Thus, 0^{th} order discontinuities of the normal buffer indicate inner edges. 1^{st} order differential operators applied to normal buffers detect them. In addition, 0^{th} order discontinuities of the normal buffer can also indicate profile edges produced by polygons that partially occlude another polygon as long as both the occluding polygon's and the occluded polygon's normal vectors are different. In this way, even profile edges producing small discontinuities in depth that are hardly detectable in the depth buffer can potentially be detected in the normal buffer. Figure 4-4 illustrates silhouette edges, border edges, and crease edges accentuating the profiles and the inner edges of the crank model.

4.3 Edge Map Construction

The *edge map* represents a 2D texture that preserves the visually important edges of 3D models as intensity values stored as texels. The process of detecting edge intensities and of constructing the edge map is based on the G-Buffer concept and the previous classification of edges.

Generation and Storage of G-Buffers

The edge-enhancement algorithm implements G-Buffers as 2D textures. To do this, it encodes the geometric properties of 3D geometries as color values using the texture's R, G, B, and A color channels. That is, after rasterizing 3D geometries, the algorithm computes the resulting fragments' geometric properties and stores the geometric properties as color values. Directing the color of each fragment to the associated texture then sets up the G-Buffer contents. To ensure a 1-to-1 relationship between pixels and texels for subsequent use of G-Buffers, both the original frame buffer and the texture are of equal size[3]. Furthermore, the algorithm exploits the

---

[3] Current graphics hardware supports *non-power-of-two* textures. Otherwise, resizing the texture to fit the next power of two for G-Buffer generation and setting up its texture matrix for accessing it later on applies as well.

graphics hardware's render-to-texture capabilities (Sec. 3.3) to generate the G-Buffer's contents efficiently.

Both the normal buffer and the depth buffer are required for edge detection. So, the algorithm must compute the fragments' normal vectors and depth values to generate these buffers. A texture's color channel generally ranges in the interval [0,1] but each component of a normalized normal vector lies within the interval [-1,1]. Thus, normal vectors must be encoded appropriately before storing them as texture values. Accordingly, the depth values that denote the fragments' distance from the camera must match the interval [0,1]. Compared to the projective depth, the depth values resulting from a linearized space ensure that the accuracy of discontinuities in the depth buffer remains independent of the 3D geometry's position and orientation in 3D space. So, it is meaningful to linearize the space defined by the 3D scene's front and the back clipping plane for calculating depth values.

To determine normal vectors and depth values on a per-fragment basis, the algorithm can either texture 3D geometry with a normalization cube map and a 1D gradient texture aligned to the viewing direction [52][95], or it can employ vertex and fragment shaders to compute them. The fragment's normal vector then populates its R, G, B color components and the depth value populates the A component of the fragment's color. As a result, a single texture can implement both G-Buffers by storing the four scalar values of the geometric properties in its R, G, B, and A channels; we refer to that texture as $T_{G\text{-}Buffer}$.

Listing 4-1 illustrates the fragment shader that computes the fragments' encoded normal and depth values for populating texture $T_{G\text{-}Buffer}$.

Listing 4-1: **Fragment Shader for Generating G-Buffers.** Encoded normal vectors and linearized depth values form the texture $T_{G\text{-}Buffer}$.

```
varying vec3 perVertexNormals;    // Interpolated per-vertex normals
varying float depth;       // Interpolated linearized depth in [0,1]

void main (void) {
    // Normalized per fragment normals
    vec3 normal = normalize(perVertexNormals);
    // Encoding normals [-1,1] => [0,1]
    normal = (normal+1.0)*0.5;

    // Output color and depth
    gl_FragColor = vec4(normal, depth);
    gl_FragDepth = gl_FragCoord.z;
}
```

Detecting Edge Intensities

The edge-enhancement algorithm detects 0^{th} order discontinuities in each G-Buffer to obtain intensity values that constitute the 3D geometries' visually important edges. The algorithm applies two 1^{st} order discontinuity operators in image space: one operating on the normal buffer and the other operating on the depth buffer.

For this, the algorithm renders to texture a screen-aligned quad covering the whole viewport of the canvas using texture $T_{G\text{-}Buffer}$ as input (Sec. 3.4). Each fragment resulting from rasterizing the quad gets assigned texture coordinates that correspond to the fragment's targeted pixel position.

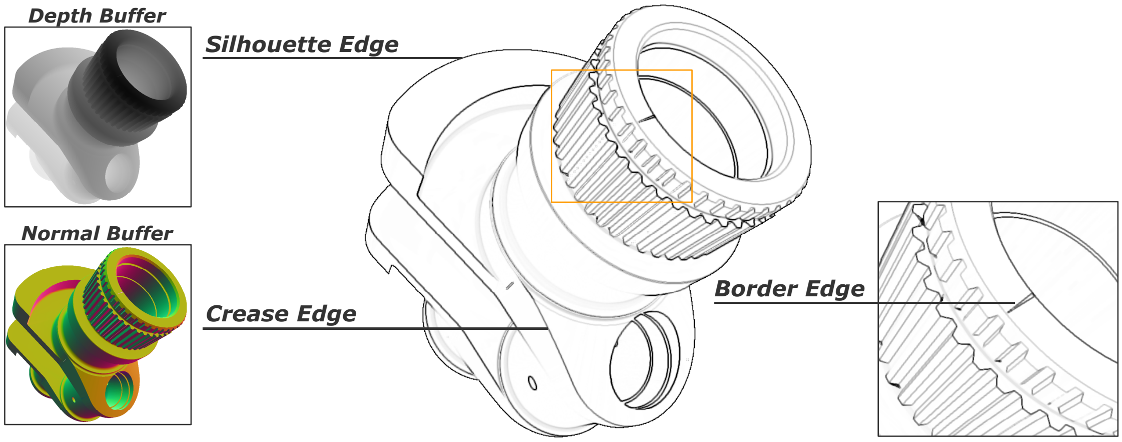


Figure 4-4: **Discontinuities in the G-Buffers Constitute Visually Important Edges.** Extracting discontinuities in the depth buffer and normal buffer result in intensity values that represent silhouette edges, crease edges, and border edges and that constitute the edge map.

In this way, fragments can access the geometric properties that have previously been generated at their pixel locations.

In a similar approach to that suggested by SAITO AND TAKAHASHI, the algorithm presented here samples neighboring texture positions to detect discontinuities in image space. But here the edge-enhancement algorithm employs image-processing operations on graphics hardware (Sec. 3.4). Furthermore, the algorithm accesses just four neighboring texture positions of $T_{G\text{-}Buffer}$ by shifting the fragments' texture coordinates slightly in A, C, F, and H direction (see Figure 3-3b). Bilinear filtering is used to access adjacent texture samples (see Figure 3-3c) in order to determine texture values as a weighted average of the four neighboring texels' values. The edge-detection operators then compare the results of two diagonally opposing texture sample pairs (A, H) and (C, F) to determine abrupt changes in the G-Buffers' contents and then accumulate the amount of discontinuity. For detecting abrupt changes in the depth buffer the algorithm evaluates the texture samples' alpha values as follows:

$$I_Z = \left(1 - \frac{1}{2}\left|A - H\right|\right)^2 \cdot \left(1 - \frac{1}{2}\left|C - F\right|\right)^2$$

Here, I_Z denotes the intensity of the discontinuity in the depth buffer. Thus, I_Z basically indicates the intenseness of the 3D model's profile edge. For detecting abrupt changes in the normal buffer the R, G, B color values are compared as follows:

$$I_N = \frac{1}{2} \cdot \left(\left(\text{normal}\left(A\right) \text{dot normal}\left(H\right)\right) + \left(\text{normal}\left(C\right) \text{dot normal}\left(F\right)\right)\right)$$

Here, the function `normal` decodes a texture sample's color value to obtain the normal vector expanded to the interval [-1,1]. Since a texture value results from bilinear filtering, normalizing the interpolated normal vector is required as well. The *dot product* then corresponds to the cosines of the angles between two opposing normal vectors. The resulting average I_N denotes the intensity of the discontinuity in the normal buffer and thus indicates the intenseness of the 3D model's inner edges.

Finally, the algorithm directs the intensity value I_Z to the R-channel, the intensity value I_N to the G-channel, and the product ($I_Z I_N$), that is, the combination of profile and inner edges, to the B-channel of the targeted texture. As a result, the assembly of edge intensities forms a single texture, called the *edge map*.

Listing 4-2: **Fragment Shader for Edge Map Construction.** Sampling neighboring positions in $T_{G\text{-}Buffer}$ enables one to detect discontinuities in both G-Buffers.

```
uniform sampler2D gBuffers; // 2D texture TG-Buffer
uniform vec2 texOff;        // Texture offset for neighbor access
uniform vec2 dim;           // Texture's/canvas's width and height

void main (void) {
    // Center and offset
    vec2 ctr = vec2(gl_FragCoord.x/dim.x,gl_FragCoord.y/dim.y);
    vec2 off = vec2((1.0/dim.x)*texOff.x,(1.0/dim.y)*texOff.y);

    // Access in direction A
    vec4 tex = gl_TextureMatrix[0]*vec4(ctr.x-off.x,ctr.y+off.y,1.0,1.0);
    vec4 A = texture2D(gBuffers, tex.xy);
    A.xyz = normalize((A.xyz*2.0)-1.0);
    // Access in direction C
    tex = gl_TextureMatrix[0] * vec4(ctr.x+off.x, ctr.y+off.y, 1.0, 1.0);
    vec4 C = texture2D(gBuffers, tex.xy);
    C.xyz = normalize((C.xyz*2.0)-1.0);
    // Access in direction F
    tex = gl_TextureMatrix[0] * vec4(ctr.x-off.x, ctr.y-off.y, 1.0, 1.0);
    vec4 F = texture2D(gBuffers, tex.xy);
    F.xyz = normalize((F.xyz*2.0)-1.0);
    // Access in direction H
    tex = gl_TextureMatrix[0] * vec4(ctr.x+off.x, ctr.y-off.y, 1.0, 1.0);
    vec4 H = texture2D(gBuffers, tex.xy);
    H.xyz = normalize((H.xyz*2.0)-1.0);

    // Calculate discontinuities
    vec3 discontinuity = vec3(0.0, 0.0, 0.0);
    discontinuity.x = 0.5 * (dot(A.xyz, H.xyz) + dot(C.xyz, F.xyz));
    discontinuity.y = (1.0-0.5*abs(A.w-H.w)) * (1.0-0.5*abs(C.w-F.w));
    discontinuity.z = discontinuity.x*discontinuity.y;
    // Output edge intensities to the edge map
    gl_FragColor = vec4(discontinuity, 1.0);
}
```

Figure 4-4 illustrates the depth buffer, the normal buffer, and the resulting edge map. Once constructed, the edge map serves as a rendering component for augmenting various rendering techniques. Furthermore, both the 3D model's profile and inner edges can be handled differently.

Listing 4-2 illustrates the fragment program for implementing edge detection and consequently edge map construction.

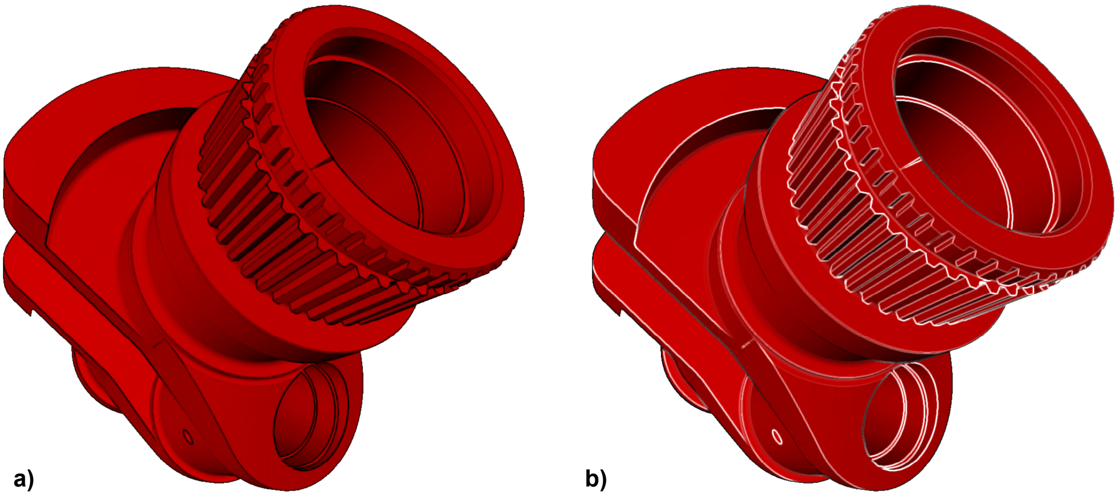


Figure 4-5: **Enhancing Edges of the Crank Model.** (a) Edge intensities of the edge map allow one to enhance the model's profile and inner edges using black. (b) Considering the edge map's edge intensities differently allows one to enhance the model's profile edges in black and inner edge in white.

4.4 Applying the Edge Map

The edge map can be used to enhance the 3D model's visually important edges in a subsequent rendering pass. For this, the intensity values of the edge map are superimposed on the 3D geometry's surface in the fragment processing stage. That is, the edge-enhancement algorithm makes the edge map, which represents a 2D rendering result produced by image-processing operations performed on graphics hardware, available in the rendering pipeline by projecting it onto 3D geometry. A fragment shader determines the texture coordinates of each of the 3D geometry's fragments in such a way that they correspond to the canvas coordinates of their targeted pixel position. The shader samples the edge map to access edge intensities and then considers the intenseness of the edges for blending between a certain edge color and, for instance, the 3D geometry's surface color. Figure 4-5a illustrates a simple shaded crank model with edges enhanced by the color black. Alternatively, the fragment shader can choose different edge colors based on the distinct edge intensities I_Z and I_N contained in the edge map. Figure 4-5b illustrates the edge enhanced crank model with black profile edges and white inner edges.

4.5 Intermediate Rendering Passes

The edge-enhancement multipass-rendering algorithm can process a set of 3D models to enhance their visually important edges. An *ordinary rendering pass* denotes a rendering pass that renders the 3D models' polygonal representation, and an *intermediate rendering pass* denotes a rendering pass that renders a single screen-aligned quad. The results of either rendering pass can be captured in 2D textures, called *intermediate rendering results*, for further use. Intermediate rendering passes perform efficiently at approximately constant cost. That is, the computational cost of an intermediate rendering pass is both independent of the number of 3D models considered for enhancement, and independent of the 3D models' tessellation. The quantity of operations processed by a fragment shader on a per-fragment basis and their individual costs as well as the canvas resolution, which affects the performance of both the render-to-texture implementation and the fill rate, are the limiting factors for intermediate rendering passes. These factors are bound to a graphics hardware performance, which is continuously increasing [63]. Thus, the constant costs of intermediate rendering passes will decrease further in the future. In conclusion, the edge-detection scheme presented here runs at a constant cost and thus will evolve along with graphics hardware evolution.

The edge-enhancement algorithm can be conceptually divided into the following three rendering passes:

1. An ordinary rendering pass is required to derive the normal buffer and the depth buffer from the 3D models.
2. An intermediate rendering pass is required to process the G-Buffers' contents and, consequently, to construct the edge map.
3. A second ordinary rendering pass is required to superimpose the edge map's edge intensities using surface shading to enhance 3D models and thus to implement the desired, e.g., non-photorealistic, style.

4.6 Applications of Real-Time Edge-Enhancement Rendering

This section presents diverse applications of the edge-enhancement algorithm.

Edge-Enhanced Technical Illustrations

GOOCH ET AL. introduce technical illustrations to demonstrate the structural as well as the material compositions of mechanical parts [45]. The enhancement of profile edges and inner edges by different edge colors is a method commonly used to increase perception in technical illustrations (Sec. 2.2).

The fragment shader in Listing 3-2 (Sec. 3.2) implements the lighting model for technical illustrations based on the Phong shading model. In addition, the fragment shader accesses the edge map to handle profile edges and inner edges differently: for inner edges, the shader linearly interpolates between the surface color determined by the lighting model and white using the edge map's intensity value I_N as weight. Then, it linearly interpolates between the color value that results from the previous weighting and a black edge color using the intensity value I_Z as weight to enhance the profile edges. Figure 3-2 depicts the resulting edge-enhanced technical illustration of the crank model.

Edge-Enhanced Cartoon Shading

Cartoon shading (a.k.a. *cel shading*) represents a typical application of edge enhancement in real-time non-photorealistic rendering [21][22]. Traditionally, cartoon style depictions feature 3D models using a reduced amount of visual detail [119]. A limited number of cartoon shades (about two or three) are used to depict a model's material color. In contrast to the method of interpolating shades smoothly across a model's surface, cartoon shades form solid, distinct color patches with respect to an incident light direction vector: a darkened version of the original material color typically depicts shadowed regions and a lightened version depicts lit regions. Sometimes a third cartoon shade depicts specular highlights using an even more lightened version.

Cartoon-shading techniques usually implement color palettes that contain cartoon shades based on the ambient, diffuse, and specular coefficients of both the model's material properties and the properties of a single light source. LAKE ET AL. [72] implement cartoon shading by utilizing a single 1D texture that produces solid color patches with sharp color transitions across the model's surface. The dot product between the normal vector at a point on the model's surface and the incident light direction vector at that position indexes the cartoon shades in the 1D texture using nearest texture filtering. On the one hand, nearest filtering facilitates the sharp

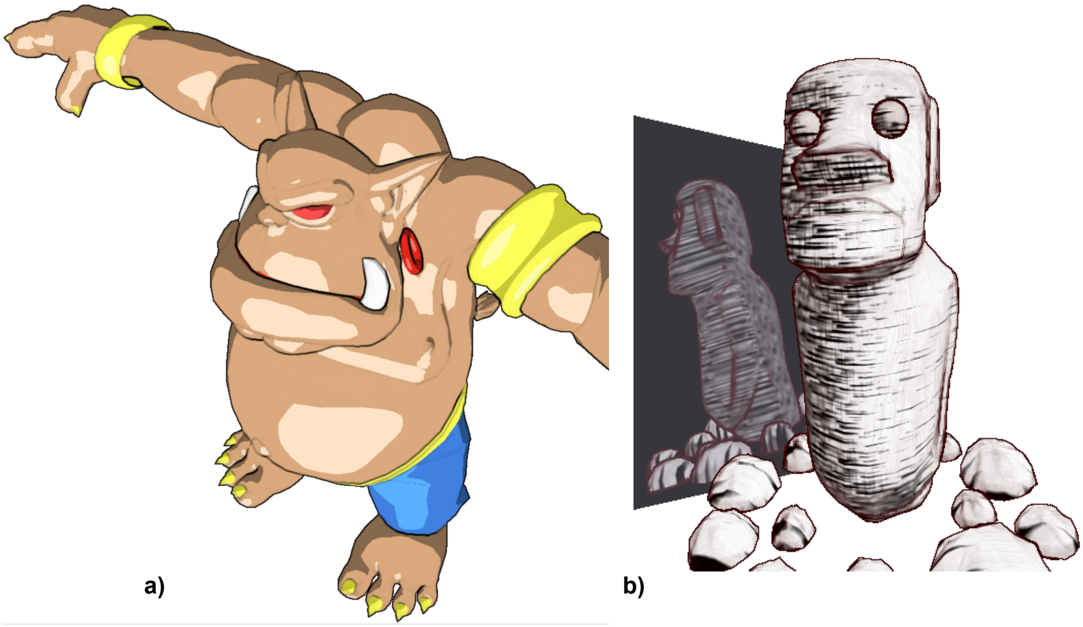


Figure 4-6: **Augmenting Non-Photorealistic Rendering Algorithms.** The edge-enhancement technique allows one to augment various non-photorealistic rendering algorithms, such as (a) cartoon-shading techniques and (b) real-time hatching techniques.

color gradients needed for cartoon shading. On the other hand, nearest filtering produces a jagged transition between consecutive color transitions [15].

Enriching cartoon shading by visually important edges is a straightforward task. Slightly modifying the algorithm presented here also allows one to reduce the visual artifacts that occur between consecutive color patches. The variation is based on multiple render targets (Sec. 3.3) and depth-sprite rendering (Sec. 3.7). In addition to normal vectors and depth values the G-Buffer generation procedure directs the cartoon-shaded color values of the model's fragments to an additional texture, called the *cartoon map*. The modified algorithm then renders a depth sprite with both the cartoon map and $T_{G\text{-}Buffer}$ as inputs. Besides constructing the edge map, it samples the cartoon map multiple times in order to blur its contents. Combining both the edge intensities and the blurred color values then gives a cartoon-shaded depiction of the 3D model that doesn't show jaggies at the color transitions (see Figure 4-6a).

Edge-Enhanced Real-Time Hatching

Stylized digital halftoning using hatching strokes represents a common technique in non-photorealistic rendering (see Chapter 2). PRAUN ET AL. and FREUDENBERG ET AL. both introduce real-time rendering algorithms for hatching arbitrary 3D models. The resulting images appear "hand made" but still provide visual cues, for instance, by conveying illumination and curvature, that let viewers perceive the surface of a 3D model. PRAUN ET AL. provide hatching strokes as textures called *tonal art maps* (*TAMs*) and apply them when shading the model's surfaces [100] (see Figure 2-1). In a similar way, FREUDENBERG ET AL. [40] blend *prioritized stroke textures* [126] with the model's surface using a *smooth threshold scheme* to implement digital halftoning facilitating various real-time NPR styles. Their approach operates on a per-fragment basis as well. Both techniques can benefit from the edge map as texture when shading the model's surface. In conclusion, a variety of edge-enhanced NPR styles can be enriched. Figure 4-6b illustrates real-time edge-enhanced hatching based on prioritized stroke textures and the smooth threshold scheme.

Applications to “Advanced Read-Time Rendering”

The term *Advanced Real-Time Rendering Techniques* [10] denotes commonly used real-time rendering techniques, such as shadowing, mirroring, and bump mapping, which are used to generate realistic depictions of 3D scenes. These techniques usually render a 3D scene or parts of it multiple times, and often exploit the graphics hardware frame-buffer resources. As a result, the implementation of these techniques tends to be complex and it is even more complex to combine them efficiently [64].

The edge-enhancement technique can be easily combined with many advanced, multipass real-time rendering algorithms. For example, the mirroring rendering technique utilizes the stencil buffer to mask a planar mirror surface in screen space. Two rendering passes are then required for scene composition: one rendering pass to render the non-mirrored view of the 3D scene and one rendering pass to render the mirrored view to fill in the stenciled region. In order to apply visually important edges in both parts of the scene, the edge-enhancement algorithm needs to be processed twice: once for the non-mirrored view and once for the mirrored view. In this way, the profile and inner edges can be extracted and enhanced correctly for both views. Figure 4-6b depicts a mirrored, edge-enhanced scene.

In general, edge map construction and edge map application are orthogonal to manifold real-time rendering techniques because only a single texture needs to be created and applied thereafter. This entire process is fully mapped to the programmable rendering pipeline. Furthermore, graphics resources, such as the stencil buffer, remain unused and thus can still be shared among advanced real-time rendering techniques.

In conclusion, the edge-enhancement algorithm can serve as a rendering component for implementations of real-time rendering algorithms even in a multipass scene graph environment [29]. For this, a multipass evaluation strategy has been implemented that applies to just a single sub graph of the scene graph [64]. In this way, manifold rendering techniques can directly benefit form the edge-enhancement algorithm.

4.7 Conclusions

The edge-enhancement algorithm is stable and robust because it can extract visually important edges in image space and only requires minor requisites, such as per-vertex normal vectors to generate the normal buffer. The algorithm does not depend on any pre-calculated information about the topology or the geometry of the 3D models. The edge-enhancement algorithm runs in real-time by exploiting current graphics hardware capabilities; it is fully integrated into the programmable rendering pipeline and scales well with hardware evolution.

The model of the crank in Figure 4-4 consists of 100.000 triangles and runs at 69.2 frames per second at a window resolution of 512×512, and at 39.4 fps at a resolution of 1024×1024 on a GeForce 6800GT graphics card. The same model consisting of 25.000 triangles takes 171.9 fps or 59.9 fps respectively.

A couple of similar real-time rendering techniques now exist that can detect edges by operating on the G-Buffer contents. Most of these techniques operate on a per-scene basis in a post-process [84]. A significant benefit of the algorithm presented here is that it preserves the edges of a designated set of 3D models as an edge map for subsequent enhancements on a per-object basis. This has the following advantages: (1) The per-object approach allows one to compose scenes that contain both edge-enhanced and non-edge-enhanced 3D models (e.g., for highlighting 3D models by accentuating just their profile edges). (2) The edge map allows one to handle edges of different 3D models individually. (3) Edge detection and preservation of edges using textures is orthogonal to most real-time rendering techniques. So, the edge-enhancement algorithm allows one to incorporate visually important edges without overusing the limited rendering resources of graphics hardware. (4) The edge map allows one to operate subsequently on the 3D models' visually important edges, e.g., to implement sketchy drawing (see Chapter 6).

A significant shortcoming of the per-object approach occurs if both edge-enhanced and non-edge-enhanced 3D models intersect one another in a single scene composition. In this case visually important edges for their distinctive display cannot be extracted directly. A solution is to take the complementary set of 3D models into account for G-Buffer generation. A scene graph [29] can explicitly declare both kinds of scene objects to configure the G-Buffer generation procedure. In this way, discontinuities can be extracted to display these 3D models distinctly.

Optimization Strategies

The color channels of a 2D texture generally have limited resolution. It can be observed that the R, G, B color components are sufficient to detect discontinuities in the normal buffer. In contrast, the resolution of the alpha channel is sometimes less adequate to detect discontinuities in the depth buffer, in particular if only minor changes in depth are present. To cope with the low precision, the near and far clipping planes of the 3D scene can be adjusted to match the bounding volumes of the set of 3D models. Additionally, either a high-precision texture (Sec. 3.3) can be used or one can capture and use the first ordinary rendering pass' z-buffer contents as a high-precision depth texture (depth map). In the case of the z-buffer's z-values, a fragment shader can compute the depth values of a linearized space. Capturing a render target's high-precision contents in a high-precision texture slightly decreases the performance of the algorithm presented here because more data needs to be processed.

The edge map can be considered just as a conceptual element for edge enhancement. The intermediate rendering pass for constructing the edge map can generally be substituted. The edge intensities can be determined in the second ordinary rendering pass while processing the 3D models' fragments. For this, the texture coordinates of the 3D models' fragments can be calculated and shifted in a similar way in order to sample the neighboring texture positions in the texture $T_{G\text{-}Buffer}$. So, a screen-aligned quad is not essential. An *early z-reject* can restrict edge-detection operations to only those fragments that contribute to the final image. That is, capturing the first ordinary rendering pass' z-buffer contents as texture allows one to test and probably reject fragments of the second pass prior to shading them. For this, the fragment shader of the second rendering pass tests the fragment's z-value with the associated z-value of the texture and possibly rejects the fragment. In conclusion, both edge detection and edge enhancement can be merged to just a single rendering pass. Nevertheless, encapsulating the edge-map construction procedure by a self-contained rendering pass reduces the complexity of the shading calculations that needs to be performed in the second pass. Constructing the edge map and reusing it as a single component thus allows one to concentrate on the implementation of the individual surface shading.

As already illustrated by the cartoon-shading example, depth sprite rendering can replace the second ordinary rendering pass

- if just the 3D models' visually important edges are to be depicted or
- if intermediate rendering results have already been produced and suffice to generate an edge-enhanced depiction, e.g., by using deferred shading based on the G-Buffer contents (Sec. 3.6).

For this, the z-buffer is once more required as texture. A depth sprite (Sec. 3.7) uses the edge map and optional auxiliary rendering results as input. In this way, one can avoid rendering the 3D models twice by simply rendering a screen-aligned quad. This can increase the performance

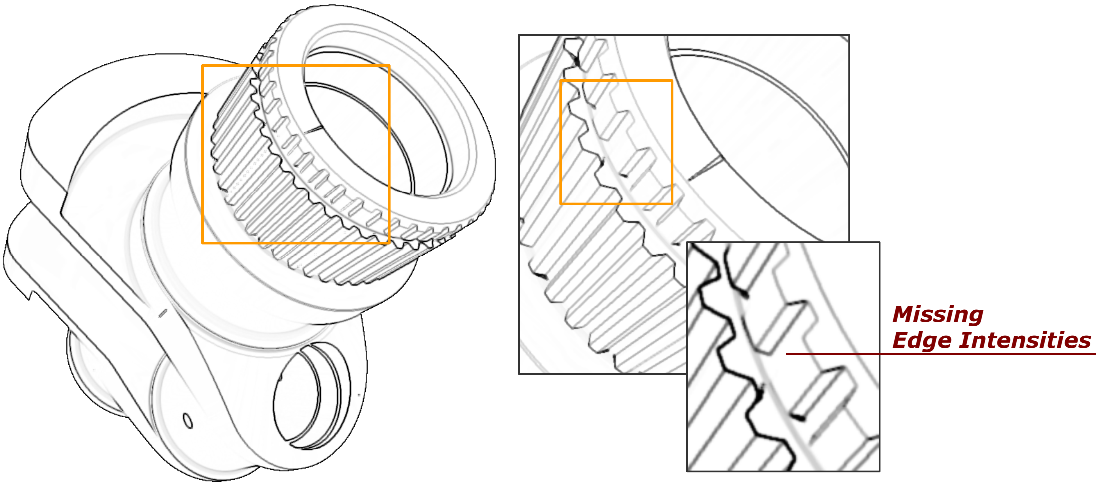


Figure 4-7: **Missing Edge Intensities.** The depth and normal buffers are not sufficient to extract the edge intensities of all visually important edges.

of the edge-enhancement algorithm significantly if the 3D models' geometric complexities are excessive.

Discussing Visual Results

The edge-enhancement algorithm samples the G-Buffers' contents at only four diagonally opposing locations around the center X (see Figure 3-3b) to determine texture values for detecting discontinuities. Other implementations consider all its neighboring texels [21][39] and apply the Soble Filter. It has already been observed that sampling four times is sufficient to generate visually pleasing results (see Figure 4-7). For comparison, NIENHAUS AND DÖLLNER [95] implement the Sobel Filter to detect discontinuities in the depth buffer. They sample texture values A through H using three intermediate rendering passes. Each of these passes samples the texture $T_{G\text{-}Buffer}$ a maximum of four times (due to the restrictions of graphics hardware at that time) using nearest texture filtering. They then apply a per-fragment blending operation to accumulate the intermediate results of a filter kernel's computation and to store the final edge intensities in the frame buffer[4]. The edge-enhancement algorithm presented here produces even more anti-aliased edge intensities compared to the result produced by the Sobel filter. The reason is that bilinear filtering is used to sample the texture $T_{G\text{-}Buffer}$ resulting in a weighted average value for the samples A, C, F, and H (see Figure 3-3). A subsequent comparison of the diagonally opposite values generates more softened edge intensity values for the edge map.

Image-space edges are approximately a few pixels wide regardless of the canvas resolution that is used. The range in which discontinuities can be detected around a texel's position can be adjusted by scaling an image-space offset for shifting texture coordinates appropriately. In this way, the edge-enhancement algorithm can increase or decrease the width of visually important edges slightly. Again, bilinear filtering preserves the visual quality of edges having an adjusted width for lower or even higher resolution canvases.

A drawback of the per-object approach presented here is that visual artifacts occur at the model-to-background boundary when the edge map is mapped onto the 3D models. The reason is that only those fragments produced for the 3D models can be used as a basis for accessing the edge

---

[4] FREUDENBERG, comes up with a technique for implementing either the horizontal or vertical part of the Sobel Filter on graphics hardware using just four texture samples [39].

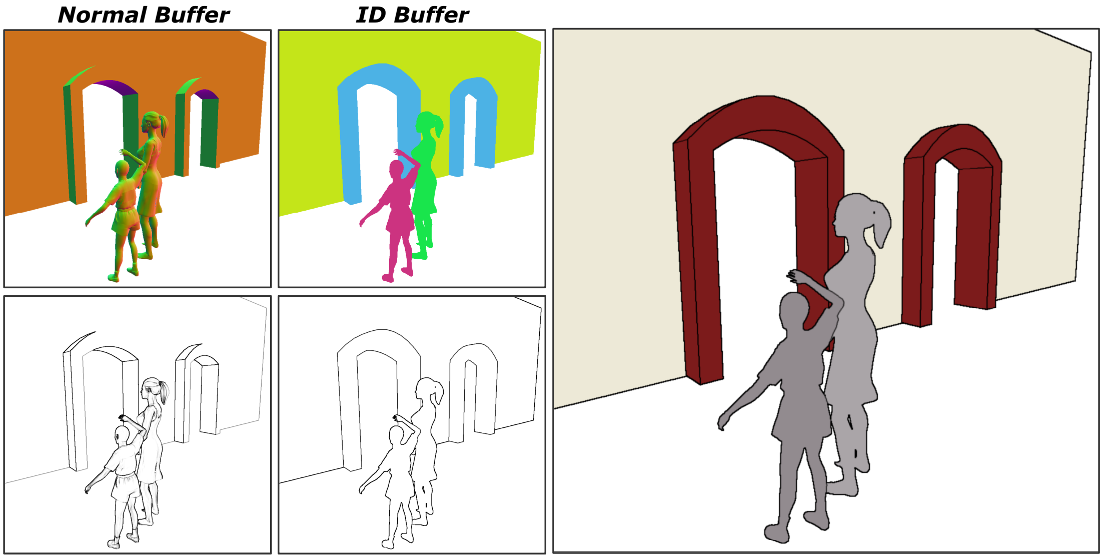


Figure 4-8: **Detecting Edges using the ID Buffer.** Detecting discontinuities in the id buffer allows one to enhance the model-to-mode or model-to-background boundary, and to display individual 3D models in a visually distinct way. The final depiction illustrates a 3D scene in which the couple is enhanced by edge intensities that have been detected in the id buffer only.

intensities, but discontinuity operators produce edge intensities near the model's boundary in image space as well. So, approximately half the width of the edges at the boundary of the 3D models gets clipped. As a result, sharp edges occur at the transition between the 3D models' image and the background. Implementing edge-enhancement as depth sprites and considering the neighboring z-values of the high-precision depth texture beyond the 3D models' boundary can avoid clipping artifacts.

Necessity for Considering the ID Buffer

Sometimes neither the depth buffer nor the normal buffer are sufficient to extract visually important edges because their contents do not show any notable changes of geometric properties although edges are present (see Figure 4-7). Usually, normal buffers show hardly any discontinuities if most of the faces in the scene are parallel to one another, e.g., when rendering artificial objects such as mechanical parts. The depth buffer barely shows abrupt changes if the scene produces a smooth depth gradient, e.g., when rendering a terrain model or an urban environment in a birds-eye-view up to the horizon. In these cases, detecting discontinuities in the id buffer (Sec. 4.1) allows one to complement profile edges by edges that indicate a model-to-model or model-to-background boundary. This allows a distinctive display of individual 3D models in images of 3D scenes. It should be noted that the id buffer cannot replace the depth buffer, because those silhouette and border edges that fall inside the 3D model's image viewed from a certain camera viewpoint cannot be detected in the id buffer.

Figure 4-8 depicts a simple scene. Here, the woman and the child are visually well-defined using the discontinuity edges of the id buffer. Abstracting from the underlying 3D model by reducing the visual complexity this way still allows one to perceive each person clearly and separately. In addition, the image shows the profile and the inner edges of the background arches. Since the depiction is visually attractive as well, graphics design decisions can be a reason for enhancing part of a 3D scene using the id buffer's discontinuity edges only, e.g., in advertisings or art.

4.8 Application to Illustrative 3D City Models

The edge-enhancement algorithm has been applied to implement *Illustrative 3D City Models* [28], which is an illustrative real-time visualization of 3D city models complementary to Virtual Reality visualizations. Illustrative 3D city models aim to accomplish the following:

- Concentrate on illustrative, expressive visualizations emphasizing high perceptual and cognitive quality that effectively communicates the contents, structure, and relationships of urban objects as well as related thematic information.
- Enable meaningful visualizations even in the case of scarce urban spatial information since high-quality and complete data is rarely available for large-scale urban areas.
- Enable the fully automated generation of visualizations while offering flexibility in controlling its graphic design.
- Achieve real-time rendering and thereby allow interactive manipulation, exploration, analysis, and editing of 3D city models.

Applications of illustrative 3D city models are primarily all of the various visual interfaces to urban spatial information required, for instance, in architectural drawings and sketches, city development planning and city information systems, the visualization of demographic development data, interactive gaming environments, and comic worlds and atmospheric environments for narratives using storyboard-like depictions (see Chapter 7).

Compositional Aspects of 3D City Environments

Buildings are the basic components of 3D city models. In general, the digital data of buildings can be acquired based on administrative data (e.g., cadastre records), laser scanning, and aerial photography. In practice, for large areas of 3D city models no explicitly modeled buildings are available. For this reason, a building's geometry has to be generated automatically. Buildings can be constructed as simplified block models by extruding 2D ground polygons to certain heights.

Environmental components include all kinds of spatial objects that set up the environmental space of a 3D city model. Examples are the following:

- Transportation networks (roads, rail, etc.)
- City furniture (street lights, advertising boards, etc.)
- Vegetation objects (trees, lawns, etc.)
- Population and traffic objects (people, cars, etc.)

Most environmental components can be modeled and handled as additional 3D scene geometry. For geometric modeling 2D polygons define the basement on top of a terrain model that represents roads, sidewalks, lawns, streets, etc. Extruding these 2D polygons generates the relevant 3D geometry.

Thematic information, e.g., demographic or building information, associated with components is defined and required by applications of 3D city models. Building information include, for instance, occupancy, industrial/residential usage, year of construction, state of restoration etc.

The visualization technique supports the mapping of thematic information to appearance parameters, that is, it "maps invisible properties onto visible attributes" [115]. The technique aims to communicate visually significant parameters of buildings and the inherent properties of buildings specified by thematic information, such as the number of floors, planning state, architectural style, etc.

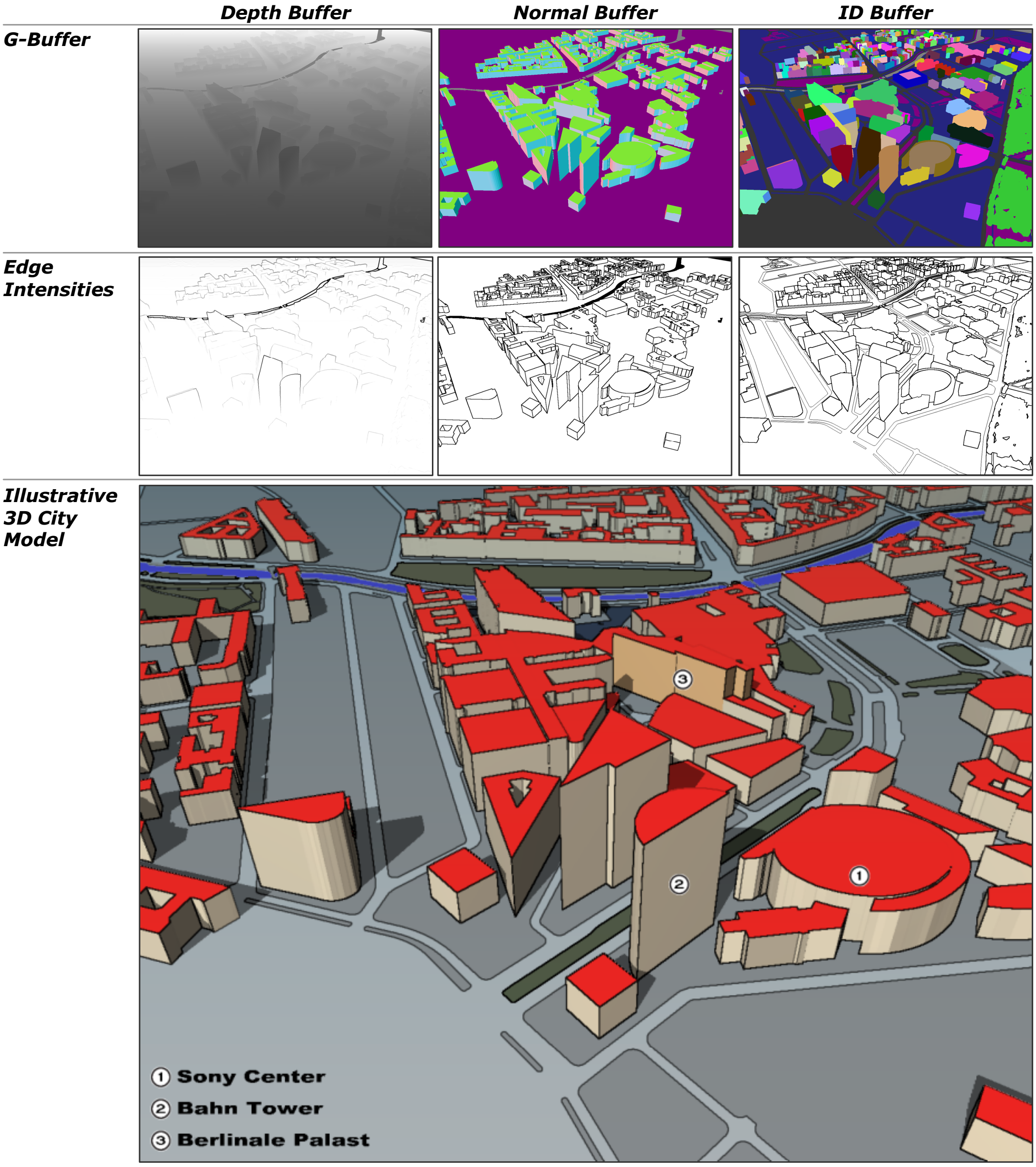


Figure 4-9: **Edge Enhancement of 3D City Models.** Discontinuities in the depth buffer, the normal buffer, and the id buffer constitute the edges intensities for enhancing 3D city models. These edges assist a homogenous and a generalized display of the 3D city model (captured from a slightly different viewpoint).

Rendering Aspects for Illustrative 3D City Models

Illustrative 3D city models apply the following depictions strategies to give a meaningful visualization.

Shadowing

Shadows in 3D city models are important cues that facilitate the perception of spatial coherence through the image. In order to calculate shadows, the real-time implementation [35] of the shadow volume technique [16] is used. Shadow volume geometry can be computed for given

3D building geometries in a pre-processing step. The shadow volume geometry is then rendered together with the 3D scene geometry to generate the stencil value zero for lit areas and non-zero for shadowed areas. Finally, the shadow information is captured in the stencil buffer as an intermediate rendering result by copying it into an alpha texture for later shadow application.

Shading of Building Geometry

Cartographic city maps and other hand drawings of cities use a reduced color scheme for shading. In colored drawings the illustrator usually simplifies the realistic colors of the urban objects and greatly reduces the number of colors used. In general, only two or three colors and only two or three tints for each color are used. The choice of the colors is based on aesthetic grounds as well as on the actual colors of the city. The visualization technique applies the *n*-tone shading introduced by DÖLLNER AND WALTHER [31] for all parts of buildings. In a similar way to cartoon-shading, the angle between the polygon's normal and the incident light direction vector provides an intensity that is used to index a color palette of *n* tones to determine the appropriate tone for shading. For this, each building is associated with a color palette defined by thematic information.

Depth Cueing

The visualization technique incorporates a depth-cueing scheme to enhance depth perception in the computer-generated depictions of 3D city models. For this, the linear transformation in *tri-stimulus color space* as described by WEISKOPF AND ERTL [122] is used. The saturation of a color is changed according to the viewer's distance from it: more distant objects are rendered in more de-saturated colors whereas the intensities of colors remain constant. In contrast to intensity depth cueing, their saturation-based depth-cueing scheme lets viewers perceive even objects in the background because their intensity contrast remains unaffected.

Facades of Buildings

Facades can encode thematic information about buildings as visual elements. To cope with large-scale 3D city models, individual textures for each building are not created. Instead, the visualization technique combines the texture elements of the facades using multi-texturing. Multiple sets of texture coordinates for each building can be created in a preprocessing step, to be used by a fragment shader that then composes the facade texture for each building procedurally.

Object-Space Edge Stylization

For object-space edge stylization the algorithm introduced by DÖLLNER AND WALTHER [31] is used. It generates quads and aligns them to visually important edges, which can easily be detected for building geometries. Texturing the quads with a stroke texture that can be partially transparent and orienting their faces towards the camera generate artistically looking stylized edges. Their shape and appearance can be individually defined to assign a characteristic appearance to visualizations and to depict specific thematic information, such as planning state or renovation state. The visualization technique can apply edge stylization to a collection of buildings at interactive frame rates.

Edge Enhancement

Image-space edge enhancement produces homogenous and generalized visual depictions of 3D building geometry while emphasizing its principle composition (see Figure 4-9 and Figure 4-10a). Since the edge map algorithm is virtually independent of the number of polygons it is capable of handling 3D scenes, which have a huge geometric complexity. Thus, image-space edge enhancement is particularly suitable for large-scale 3D city models but also applies well to

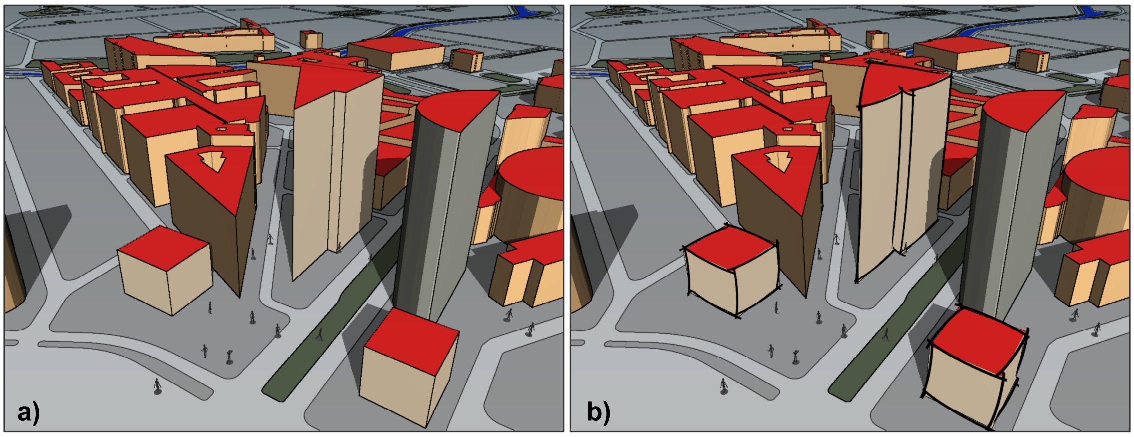


Figure 4-10: **Image-Space and Object-Space Edges for Illustrative 3D City Models.** (a) The depiction illustrates the 3D city model using image-space edges only, whereas (b) the other depiction shows additional stylized object-space edges applied to a subset of buildings

the ground geometry, which generally contains a high number of curved shapes leading to a large number of short edges.

A typical characteristic of 3D city models is that a lot of faces are parallel to one another, e.g., buildings, roofs, and basement geometry. Furthermore, the scene measured from the front-most buildings to the buildings far away near the horizon is typically large with respect to depth, especially if viewed from a birds-eye-view. As a result, the normal buffer contains identical normal values for different faces, and the depth buffer just shows a smooth depth gradient producing only minor changes in depth. These buffers are therefore not sufficient to produce adequate profile edges. For this reason, the id buffer is required. It allows one to encode each single building as well as the basement geometries by an individual color value, the object identifier. In this way, profiles can be extracted that display both building-to-building and building-to-ground distinctions and accentuate the outlines of roads, sidewalks, lawns, etc. [12].

Figure 4-9 depicts the depth buffer, the normal buffer, and the id buffer as well as the corresponding edge intensities needed for enhancing 3D city models.

Hybrid Rendering

The visualization technique for illustrative 3D city models applies a hybrid-rendering algorithm that combines the advantages of both edge-rendering approaches, that is, stylizing the buildings' edges to communicate thematic information, such as planning states, and accentuating the image-space edges to generalize principle compositions. When shading the 3D city model geometry using the shadow-alpha-texture, *n*-tone shading, and depth cue calculations one can also apply the edge map to accentuate image-space edges. Note that restricting image-space edge-enhancement to a designated set of buildings avoids interference with the objects-space stylization typically applied for the complementary set of buildings (see Figure 4-10b). Since the edge-enhancement preserves a correct z-buffer behavior, objects-space edges and any additional 3D scene geometry can easily be combined.

Figure 4-11 illustrates a part of a 3D city model. The 3D representations of people and facade textures help one perceive measuring units, i.e., one can estimate the height of the depicted buildings. Procedural facade texturing lets one also perceive the industrial or residential usage of a building, and its occupancy.

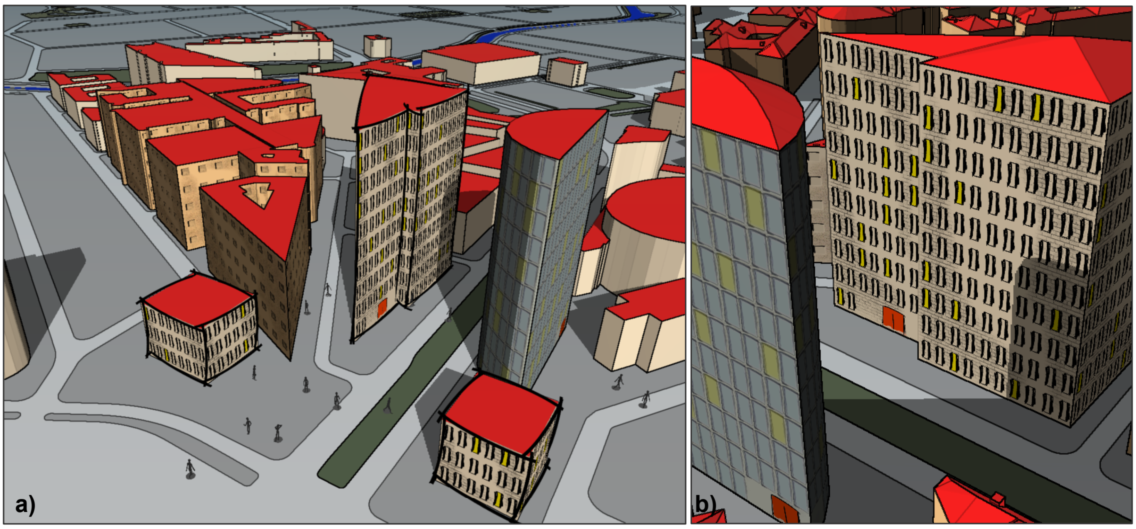


Figure 4-11: **Conveying Thematic Information in Illustrative 3D City Models.** (a) Procedurally generated facades and people encode non-geometric information about in illustrative 3D city models. (b) One building represents a modern office building; the other building represents a residential building.

The assembly of all rendering techniques provides a high degree of freedom for graphic design to illustrate 3D city models expressively and to visualize their related thematic information, e.g., in order to design city information systems. Figure 4-9, for instance, depicts a 3D city model from a bird's eye view using references that provide textual information for buildings.

Chapter 5

Blueprint Rendering Technique

Outlining and enhancing visible and occluded features in drafts of architecture and mechanical parts are essential techniques for visualizing complex aggregate objects and for illustrating the position, layout, and relations of their components. This chapter introduces blueprint rendering that enhances the *visible as well as the occluded* visually important edges of arbitrary 3D geometries [92].

The word *blueprint* in its original sense is defined by Merriam-Webster as “a photographic print in white on a bright blue ground or blue on a white ground used especially for copying maps, mechanical drawings, and architects' plans”. Blueprints consist of transparently rendered features, represented by their outlines. Thus, blueprints make it easy to understand the structure of complex, hierarchical object assemblies such as those found in architectural drafts, technical illustrations, and designs.

In a naive approach to blueprints, a wire-framed depiction could be used, but would not allow one to distinguish between polygonal edges and true outlines, such as silhouettes. This depiction even complicates the visual perception of complex object assemblies (see Figure 5-1b). One could also use transparency rendering, but outlines would hardly be visible, in particular in regions of high depth complexity (see Figure 5-1c). Image-space non-photorealistic algorithms operate only on visible features and cannot be directly extended to transparent rendering.

The blueprint rendering technique presented in this thesis extends the edge-enhancement rendering algorithm for accentuating visually important edges of 3D models (see Chapter 4) to their occluded parts [89][92]. For this, the edge map construction is combined with depth peeling, a technique that extracts disjunctive layers from the 3D geometry which represent its graphical decomposition, in order to cope with its depth complexity. Vivid and expressive depictions of complex aggregate objects become possible and facilitate visual perception (see Figure 5-1d).

Blueprint rendering serves as an effective tool for interactively exploring, visualizing, and communicating spatial relationships. Among the many application areas, blueprints can be used for visualizing and illustrating ancient architecture of cultural heritage [90]. For instance, they can help to guide viewers through dungeon-like environments, and can highlight hidden chambers and other components found in archeology, such as tombs. Furthermore, blueprint rendering makes it easy for artists and design engineers to obtain insights into complex aggregate objects and thus to ease their construction.

Constructive Solid Geometry (*CSG*), which is used to model complex, aggregate objects, represents a typical field of application of blueprint rendering. Blueprint rendering can be

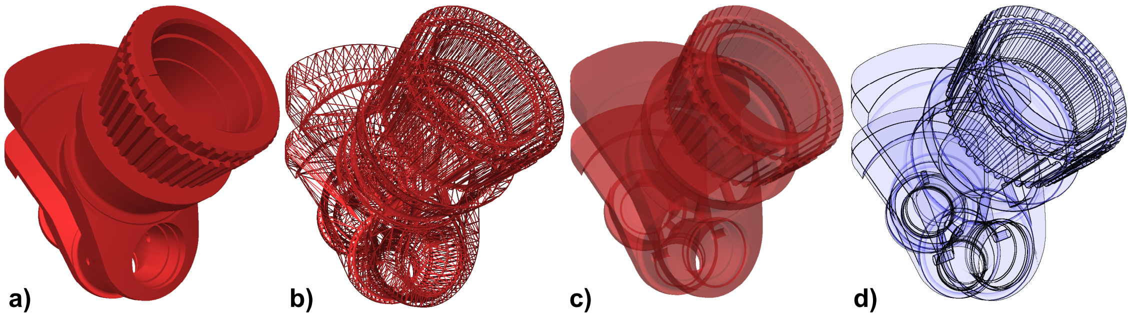


Figure 5-1: **Illustrating the Layout of a Mechanical Part.** (a) A rendering of the model of the crank. (b) A wire-framed depiction complicates our perception of the shape. (b) A transparency rendering can show its occluded parts. (d) A blueprint depiction of the model makes it easier to see layout.

seamlessly integrated into image-based CSG rendering to ease the interactive composition of CSG shapes.

The remainder of this chapter is structured as follows. Section 5.1 describes depth peeling and its adaptation to blueprint rendering. Section 5.2 presents the blueprint rendering technique. Section 5.3 presents an extension of blueprint rendering that generates edge-enhanced order-independent transparent depictions. Section 5.4 gives details of depth masking. Section 5.5 outlines applications to architectural drafts. Section 5.6 presents the extension of blueprint rendering to image-based CSG rendering. Section 5.7 draws conclusions.

5.1 Depth Peeling Technique

Depth peeling is a multipass rendering technique that operates on a per-fragment basis and extracts 2D layers from 3D geometries; these layers have a depth-sorted ordering. Generally speaking, depth peeling successively “peels away” layers of unique depth complexity.

Previous Work

Based on the *Virtual Pixel Maps* architecture, MAMMEN [75] introduces an approach to processing pixels in depth-sorted order when rendering 3D geometries. As one application, he generates an ordering of transparent pixels suitable for implementing high-quality antialiased transparency rendering. DIEFENBACH extends this approach to general hardware using the *dual z-buffer* concept. The dual z-buffer allows him to implement two depth tests on a per-fragment basis [26]. When rendering 3D geometries multiple times, the second depth test lets him process fragments in depth-sorted order to implement transparency rendering as well. Finally, EVERITT [36] introduces a hardware-accelerated solution for the dual z-buffer on common graphics cards using shadow maps [109]. He implements depth peeling by extracting layers of ordered depth to facilitate order-independent transparency rendering in real-time.

Depth Peeling Implementation

In general, the fragments that pass an ordinary depth test define the minimal z-value at each pixel. But one cannot directly determine the fragment that comes second (or third, etc.) with respect to its depth complexity. Thus, an additional depth test to extract those fragments that form a layer of a given ordinal number (with respect to depth complexity) is required. In order to eliminate fragments that have a lower depth complexity than those of a certain layer, the second depth test requires the z-values of the preceding layer. Depth peeling provides them with a texture. With depth peeling, one can thus extract the first n layers using n rendering passes.

A *depth layer* denotes a layer of unique depth complexity and a *depth layer map* denotes a high-precision texture received by capturing the associated z-buffer contents. Accordingly, a *color layer map* denotes an additional texture that captures the contents of the associated color buffer. Thus, both maps serve as intermediate rendering results that can be reused subsequently. In particular, color layer maps can later be used in depth-sorted order to compose the final rendition, e.g., for implementing order-independent transparency.

The pseudocode in Listing 5-1 outlines the implementation of depth peeling for blueprint rendering. It operates on a set G of 3D geometries. Here, G is rendered multiple times, whereby the rasterizer produces a set F of fragments. The loop terminates if no fragment gets rendered (*termination condition*); otherwise, the technique continues with the next depth layer. That is, if the number of rendering passes has reached the maximum depth complexity, the condition is satisfied.

Listing 5-1: **Depth Peeling for Blueprint Rendering.** The pseudocode illustrates the combination of both depth peeling and edge-map construction for implementing blueprint rendering.

```
procedure depthPeeling(G ← 3DGeometry) begin
  int i=0
  do
    F ← rasterize(G)
    if(i==0) begin
      /* Perform ordinary depth test in the first rendering pass */
      for all fragment ∈ F begin
        bool test ← performDepthTest(fragment)
        if(test) begin
          fragment.depth → z-buffer
          fragment.color → color buffer
        end
        else reject fragment
      end
    end
    else begin
      /* Perform two depth test */
      for all fragment ∈ F begin
        /* First depth test */
        if(fragment.depth > fragment.value$_{depthLayerMap(i-1)}$) begin
          /* Second depth test */
          bool test ← performDepthTest(fragment)
          if(test) begin
            fragment.depth → z-buffer
            fragment.color → color buffer
          end
          else reject fragment
        end
        else reject fragment
      end
```

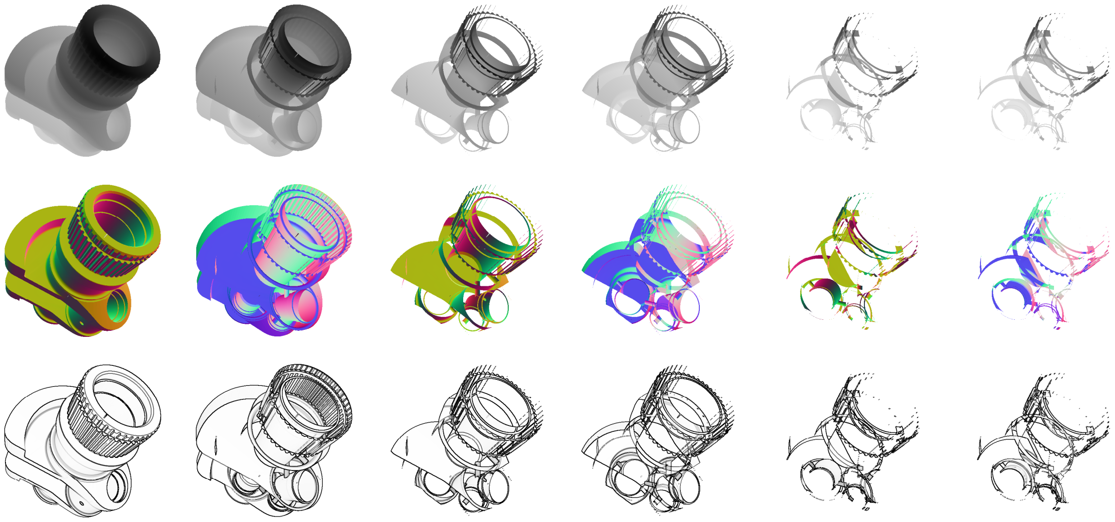


Figure 5-2: **G-Buffers and Edge Maps of Consecutive Depth Layers.** The depth buffer (first row) and the normal buffer (second row) of each depth layer (column) form the basis for constructing the edge map (third row) for each layer.

```
        end
        depthLayerMap$_{(i)}$ ← capture(z-buffer)
        colorLayerMap$_{(i)}$ ← capture(color buffer)
        /* Edge intensities */
        edgeMap$_{(i)}$ ← edges(depthLayerMap$_{(i)}$,colorLayerMap$_{(i)}$)
        i++
    while(occlusionQuery() $\neq \varnothing$ )  /* Termination condition */
end
```

Performing Two Depth Tests

In the first rendering pass ($i = 0$), depth peeling performs an ordinary depth test on each fragment. The contents of the z-buffer and the color buffer are then captured in either the depth layer map or the color layer map for further use.

In consecutive rendering passes ($i > 0$), depth peeling performs an additional depth test on each fragment. For this test, it applies the depth layer map of the previous rendering pass (i-1). Depth peeling determines the fragment's texture coordinates in such a way that they correspond to the canvas coordinates of the targeted pixel position. In this way, a texture access provides a fragment with the z-value stored at that pixel position in the z-buffer of the previous rendering pass.

Now, the two depth tests work as follows:

- If the current z-value of a fragment is greater than the texture value that results from accessing the depth layer map, the fragment proceeds and the second ordinary depth test is performed.
- Otherwise, if the test fails, the fragment gets rejected.

When all the fragments have been processed, the contents of the z-buffer and the color buffer form the next depth layer map and color layer map. A fragment shader can implement the

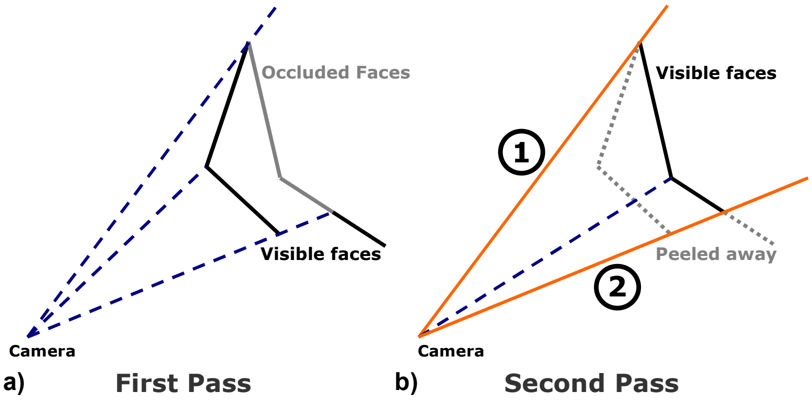


Figure 5-3: **Two Possibilities When Peeling Away 3D Geometry.** Rays are cast to discontinuities produced by the composition of polygons. They are visible from the camera position. (a) The composition of upright polygons illustrates the visible faces of the first rendering pass and (b) the system with faces peeled away in the second rendering pass. Here, the solid orange rays indicate edges that exist in both edge maps.

additional depth test efficiently, and occlusion queries (Sec. 3.1, [61][108]) can implement the termination condition efficiently.

5.2 Blueprint Rendering

Blueprint rendering extracts both the visible and non-visible edges of 3D geometry. *Visible edges* denote visually important edges that are directly seen by the virtual camera, whereas *non-visible edges* denote visually important edges that are occluded by faces of 3D geometry, that is, they are not directly seen. The blueprint rendering technique combines the edge-enhancement algorithm with the depth-peeling technique to extract these edges. Once they have been generated, visible and non-visible edges can be composed as the final blueprint depiction in the frame buffer.

Extracting Visible and Non-Visible Edges

The depth-peeling technique invokes the edge-map construction for each depth layer. Since discontinuities in the normal buffer and depth buffer constitute the visible edges, both are required for each rendering pass. For this, the blueprint rendering technique encodes the fragments' normal vectors as color values to generate the normal buffer as the color layer map. The edge map can then be constructed directly because the depth layer map already forms a valid depth buffer (Sec. 4.7) and thus can be used.

Non-visible edges become visible when successive depth layers are peeled away. Consequently, the modified depth-peeling technique can also extract non-visible edges (already shown in Listing 5-1).

As a result, the blueprint rendering technique preserves visible and non-visible edges as edge maps for further processing. Figure 5-2 shows the depth buffers, the normal buffers, and the resulting edge maps of successive depth layers.

It should be noted that visually important edges in the edge maps of consecutive depth layers appear repeatedly because local discontinuities can remain when peeling away faces of 3D geometry. Consider the following cases:

1. Two connected polygons share the same edge. One polygon occludes the other one. The discontinuity in the z-buffer that is produced along the shared edge will remain when peeling away the occluding polygon.

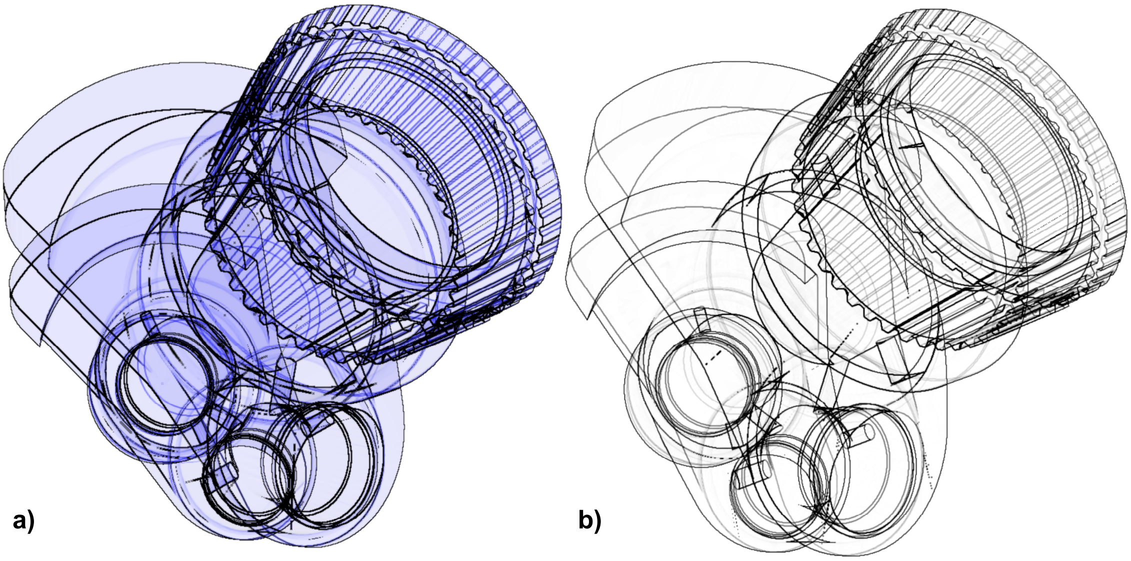


Figure 5-4: **Blueprint Rendering Provides Insights into 3D Models.** Blueprint rendering enables one to depict 3D models (a) by their visible and occluded edges and by depth complexity cues or (b) by a wire-frame like style based on visually important edges only.

2. A polygon that partially occludes another polygon produces discontinuities in the depth buffer at the transition. When peeling away the occluding polygon and non-occluded portions, a discontinuity in the depth buffer will be produced at the same location.

Figure 5-3 illustrates both cases. However, the performance of edge-map construction is virtually independent of the number of discontinuities.

Composing Blueprints

The blueprint rendering technique composes blueprints using visible and non-visible edges stored in edge maps in depth-sorted order. For each edge map, blueprint rendering proceeds as follows:

- It textures a screen-aligned quad that fills in the viewport with the edge map and its associated depth layer map as input.
- Blueprint rendering then applies a fragment program that (1) implements depth sprite rendering using the associated depth layer map; (2) calculates the fragment's *R*, *G*, and *B* color values using the edge intensity value derived from accessing the edge map and, for instance, a bluish color; and (3) sets the fragment's alpha value to the edge intensity.
- Finally, the technique uses color blending by considering the edge intensity values as blending factors to provide depth complexity cues while keeping edges enhanced.

Note that the edges that appear repeatedly in edge maps of consecutive depth layers superimpose on one another without disturbing artifacts. Figure 5-4a shows the resulting blueprint of the mechanical part. Alternatively, we can depict 3D models in a wire-frame style that is based on visually important edges only (see Figure 5-4b). For this, a threshold value is defined that rejects fragments on the basis of their associated edge intensity. Fragments that pass the threshold test then form a wire-framed depiction of the 3D geometries. In either case, composing blueprints using depth sprite rendering enables one to combine the 3D geometries' blueprint depictions with further 3D scene contents.

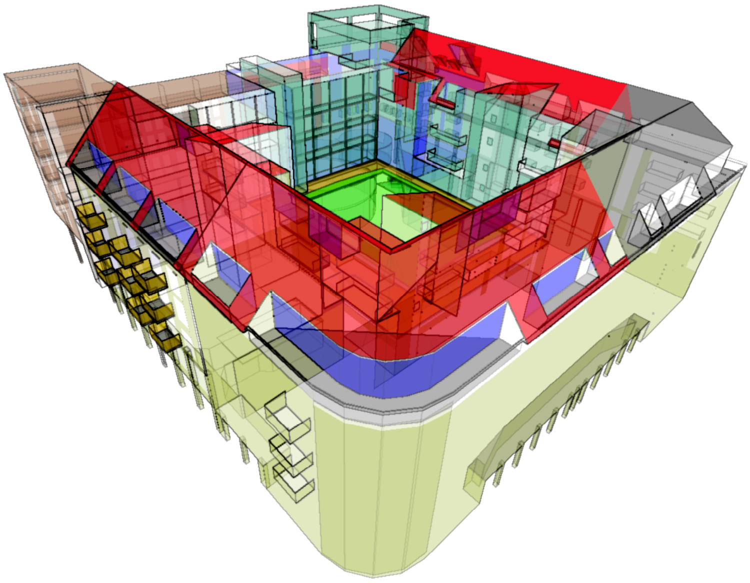


Figure 5-5: **Edge-Enhanced Order-Independent Transparency Rendering.** Blueprint rendering can be used to synthesize edge-enhanced order-independent transparency renderings of 3D models. The depictions let one perceive individual parts and sections of the architectural building, a residential building.

Performance Considerations

It can be observed that it is sufficient to blend just the first few layer maps in order to compose blueprints. The remaining layer maps have less visual impact on the overall composition because only a few (often isolated) pixels get colored. To alter the termination conditions for blueprint rendering and thus to optimize rendering performance, a desired minimal fraction of fragments (depending on the window resolution) can be specified to pass the depth test. In this way, the number of rendering passes can be decreased while maintaining a desired visual quality of the blueprints. To implement the trade-off between speed and quality, one can configure the occlusion query appropriately.

When considering five depth layers, the model of the crank in Figure 5-4a (100.000 triangles) takes 15.8 fps at a window resolution of 512×512 and 8.9 fps at a window resolution of 1024×1024 on a GeForce 6800GT graphics card. The same model with a reduced geometric complexity (25.000 triangles) takes 24.9 fps and 11.8 fps respectively.

5.3 Edge-Enhanced Order-Independent Transparency

Although blueprint depictions communicate layouts and relations efficiently by outlining the visible and occluded features of 3D geometries and by providing depth complexity cues, they do not illustrate the spatial orientation of single features very well. In contrast, the 3D geometries' surface shading, which allows one to perceive the orientation of their faces in 3D space, provides a particularly important spatial cue for vivid visualizations. Transparency rendering provides these spatial cues even for the occluded faces of 3D geometries. Thus, the combination of both blueprint rendering and transparency rendering seems to be a meaningful extension to the visualization of complex aggregate objects.

Since depth peeling was originally developed for order-independent transparency rendering, we can benefit directly from the implementation presented here. For this, two color layer maps for

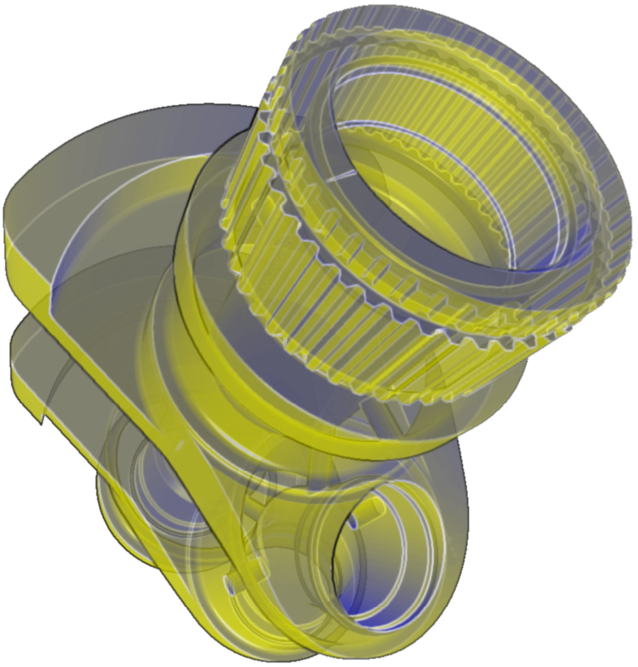


Figure 5-6: **Blueprint Rendering for Technical Illustrations.** Blueprint rendering using the Gooch lighting model for technical illustrations depicts the mechanical part, the crank.

each depth layer have to be synthesized: one color layer map to preserve the 3D geometries' surface shading and one color layer map to preserve the normal buffer. Both textures can be generated simultaneously by means of multiple render targets (Sec. 3.3). As before, each rendering pass constructs the edge map for its depth layer. The edge map and the color layer map can then be combined when rendering each depth layer as a depth sprite. The intensity values of the edge maps multiplied by the color values of the color layer map that contains surface colors form the finals colors. The rendering technique then blends each layer with the frame buffer contents using the alpha values of the color layer map as blend factors. Figure 5-5 illustrates the resulting edge-enhanced transparent depiction of an architectural building. A transparently textured model can, of course, be depicted in a similar way if this kind of effect is desired and is a reasonable way of communicating the assemblies.

Furthermore, DIEPSTRATEN ET AL. suggest applying the Gooch lighting model (Sec. 2.2 and 4.6) to generate technical illustrations for *view-dependent transparency rendering* [27] at least to some parts of an aggregation. Since visualizing the design of mechanical parts is one of the major application areas for which blueprint rendering has been developed, the Gooch lighting model can certainly be applied as well instead of a simple lighting model (see Figure 5-6).

5.4 Depth-Masking

Depth masking is a technique that peels away a minimal number of depth layers until a specified fraction of an assembly's designated occluded components, such as the row of statues in Figure 5-7, becomes visible. In fact, depth masking provides a termination condition for blueprint rendering to adapt the number of rendering passes dynamically. Depth masking proceeds as follows:

1. It captures a high-precision depth texture, called a *depth mask*, derived from rendering only the designated components in a first rendering pass.

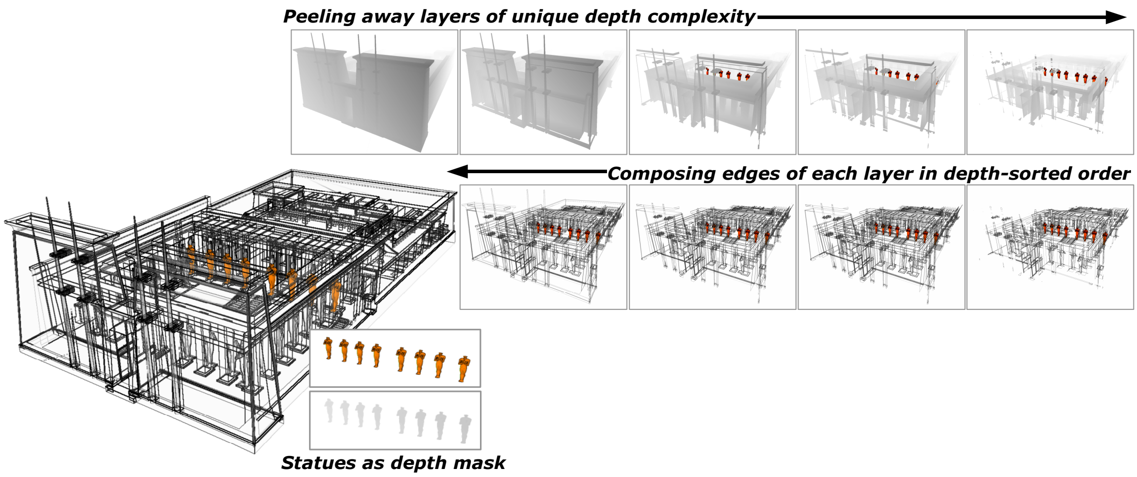

Peeling away layers of unique depth complexity
Composing edges of each layer in depth-sorted order
Statues as depth mask

Figure 5-7: **Concept of Depth Masking.** Depth masking allows one to reduce the visual complexity in blueprints by considering just a minimal number of depth layers with respect to a depth mask.

2. In successive rendering passes, the depth-masking technique renders the depth mask as a depth sprite whenever a depth layer has been peeled away. If a specified fraction of the number of fragments passes the ordinary depth test (based on the z-buffer contents just produced), the technique terminates. Otherwise, more depth layers must be peeled away.
3. Finally, the designated components can be simply integrated when composing blueprints.

The modifications to the blueprint rendering technique are shown in pseudocode in Listing 5-2. Again, occlusion queries implement the adjusted termination condition.

Listing 5-2: **Implementing the Depth Masking Technique.** The pseudocode illustrates the modified termination condition for implementing depth masking.

```
procedure depthPeeling(G ← 3Dgeometry,
                       C ← geometryOfOccludedComponents) begin
  /* Render components and caputer z-buffer as depth mask */
  depthMask ← depthTexture(C)
  quad ← createTexturedScreenAlignedQuad(depthMask)
  renderDepthSprite(quad)
  int Q = occlusionQuery()
  int i=0
  do
    ... /* Rasterize geometry, perform depth tests, capture
           layer maps, and construct edge maps (see Listing 5-1) */
    renderDepthSprite(quad)
    int R = occlusionQuery()    /* Number of visible fragments of
                                   the component */
  while(R<fraction(Q) )         /* Termination condition */
end
```

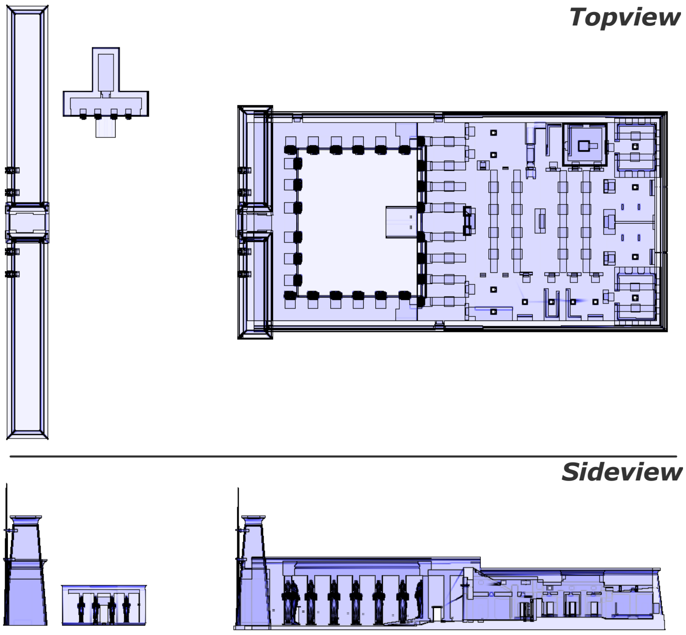


Figure 5-8: **Plan Views of the Temple of Ramses II in Abydos.** Top and side views illustrating the layout of the temple allow one to clearly identify its chambers, pillars, statues, etc.

In conclusion, depth masking can reduce the visual complexity in a blueprint depiction by considering just a minimal number of depth layers with respect to a depth mask.

Figure 5-7 illustrates the process of depth masking considering the row of occluded statues, which are part of a temple whose structural complexity is excessive.

5.5 Applications to Illustrations of Ancient Architecture

Besides illustrating mechanical parts, visualizing the architecture of building in order to explore and communicate them represents another application area for blueprint rendering [90].

Architectural Drafts as Plan Views

With blueprint rendering, one can generate architectural plan views automatically in order to provide comprehensible architectural outlines. Composing plan views using an orthographic camera for rendering is a straightforward task. Here, edges and depth complexity cueing are sufficient to differentiate single components in the overall composition. In the plan views of the *Temple of Ramses II* in Figure 5-8, chambers, pillars, and statues can be clearly identified. Thus, blueprints increase visual perception. Orthographic views are even more appropriate than perspective views if an overview is needed and if the structural complexity of the architecture is more than can be reasonably displayed.

Highlighting Occluded Components in Architectural Drafts

In contrast, a perspective view still provides better spatial orientation and conceptual insight in blueprints of architecture than orthographic views. Highlighting certain occluded components in a perspective view of an architectural composition allows one to focus on them and to understand their relation to the entire composition. Figure 5-7 illustrates the design of the entrance and the inner yard of the temple with its surrounding walls and statues. These are in front of the highlighted statues that guard the doorway to the rear part of the temple. Here, depth

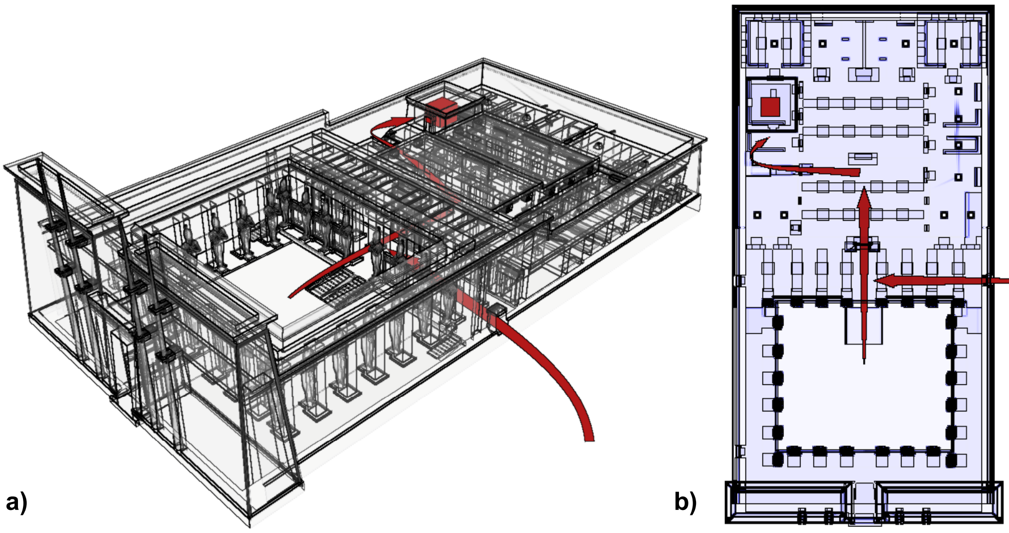


Figure 5-9: **Illustrating Locations and Directions.** Enhancing (b) a perspective and (c) an orthographic view of the Temple of Ramses II in Abydos using glyphs illustrates locations and directions and thus provides guidance in these depictions.

masking is used to peel away a minimum number of depth layers so that the occluded statues become entirely visible. Furthermore, depth masking reduces the visual complexity: in contrast to the depiction shown in Figure 5-9a, the entire complexity of the temple, in particular in the rear part, is not outlined in Figure 5-7.

Relations and Locations in Architectural Drafts

Enhancing blueprints of architecture using glyphs assists one to communicate hidden details, locations, and relations, and to guide one through the composition. For this, general 3D geometry can be integrated in blueprints to provide additional knowledge in depictions of architecture. The illustrations in Figure 5-9 mark a hidden chamber (red box) in the rear part of the temple and the pathways to guide one to the chamber (red arrows).

5.6 Applications to Illustrations of CSG Models

In interactive applications for constructing CSG models, the user requires both: a constant visual feedback to facilitate the interactive composition of various transient CSG components as well as a comprehensible visualization of the CSG models' design and spatial assembly.

CSG modeling means defining 3D geometry as a result of set operations ($\cup,\cap,-$), applied to basic, closed 3D primitives or to other CSG geometry defined in this way. The CSG tree represents the fundamental structure for specifying a CSG shape (see Figure 5-10a and Figure 5-10b).

Generating a 3D polygonal representation of a CSG shape's surface is computationally expensive, and hardly possible in real-time for complex models. Hence, in interactive applications for modeling CSG geometry, rendering CSG models using an object-space approach, e.g., by determining their boundary representation [38], is less appropriate. Today's image-based CSG rendering algorithms synthesize a graphical representation of CSG models in real-time without calculating the triangulation of their final 3D geometric mesh explicitly. GOLDFEATHER ET AL. present an algorithm for image-based rendering of CSG [41]. An important part of their work is the normalization of arbitrary CSG trees into an equivalent *union-of-partial-product* form. Visibility of partial products (a.k.a. *CSG products*) can be

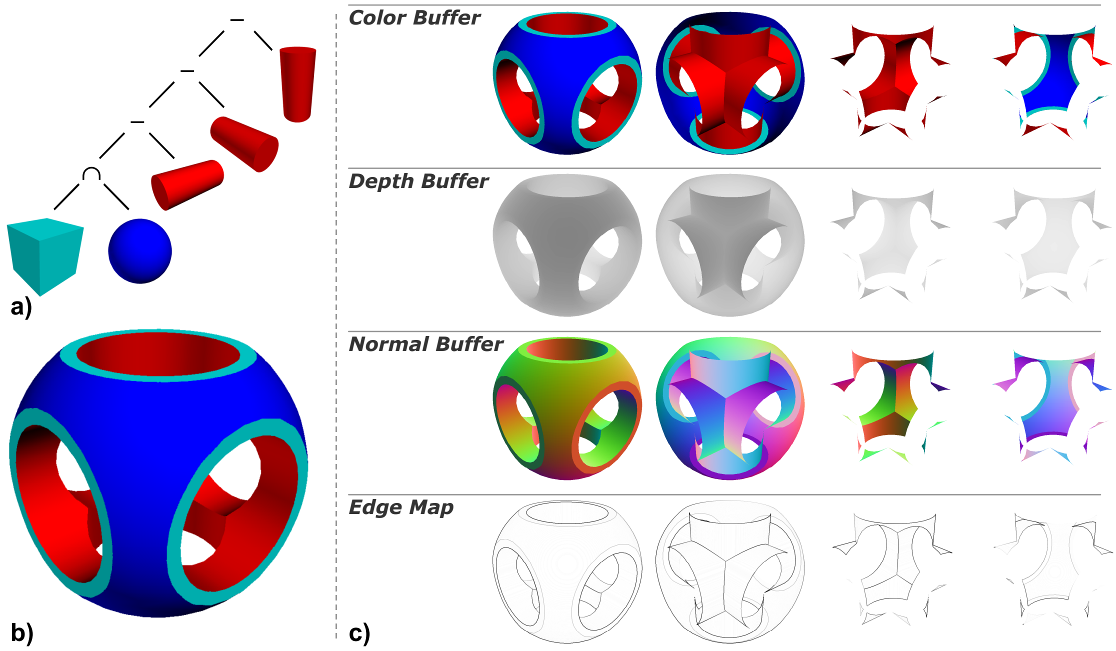


Figure 5-10: **Disjunctive Depth Layers of CSG Models.** (a) The CSG tree shows a partial-product of a sphere and a box from which we subtract cylinders. (b) This results in the CSG model of a widget. (c) The color buffer (first row) of each depth layer (column) can be used for the order-independent transparency rendering of CSG shapes. Discontinuities in the depth (second row) and the normal buffers (third row) form the edge maps (fourth row) of each depth layer.

effectively determined by image-space graphics hardware operations. Wiegand describes a rendering technique that implements the algorithm of Goldfeather et al. using OpenGL [123]. Kirsch and Döllner improve this approach by a real-time capable rendering technique that transfers visibility information using color textures [66].

Depictions produced by image-based CSG rendering algorithms generally illustrate CSG models by simple shaded geometries represented only by the outer surface of their solids (see Figure 5-10b). So, the entire assembly is difficult to understand, which makes comprehension even more difficult when modeling them. The application of blueprint rendering to image-based CSG rendering aims to produce depictions that will allow one to perceive the visible and occluded parts as a whole, in order to understand the position and orientation of the aggregated components of the CSG models. Depth peeling is thus required, but does not map directly to image-based CSG rendering due to the visibility transfer necessary for generating the CSG models' image. Guha et al. [47] implement depth peeling for image-based CSG rendering up to the second depth layer. In contrast, Kirsch presents a solution for depth peeling of CSG models that can even cope with their entire depth complexities and can extract depth layers in real-time [64]. Using a method based on his approach blueprint rendering for visualizing the design and spatial assembly of interactive CSG models can be implemented [65].

Depth Layers of CSG Models

A *front surface* denotes those surfaces of a CSG model that face towards the viewer and a *back surface* denote the surfaces that face away from the viewer. Both front and back surfaces form the closed surface of a CSG model. In common depictions of CSG models, only the nearest front surface of a CSG model (with respect to depth complexity) is visible and thus forms the

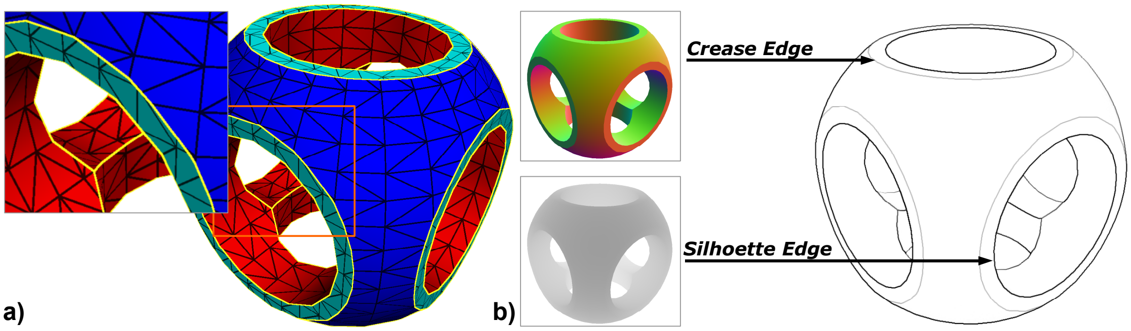


Figure 5-11: **Visually Important Edges of CSG Models.** (a) Visually important edges of a CSG model do not correspond to the edges of the polygonal mesh. (b) Based on the normal and depth buffers, the edge-enhancement algorithm allows one to detect silhouette and crease edges for image-based CSG rendering.

first depth layer. In contrast, inner depth layers of a CSG model, that is, layers that become visible when using depth peeling, can consist of both front and back surfaces.

When extracting the depth layers of CSG models, the z-buffer gets captured as the depth layer map and thus can be reused as a depth buffer. While rendering visibility information, which flags visible and non-visible parts of the CSG model, multiple color layer maps can be constructed simultaneously (using multiple render targets) [64]. In this way, rendering the surface's color values can construct a color buffer for each depth layer; rendering encoded normal vectors can construct the normal buffer for each depth layer. Figure 5-10c illustrates the color buffer, depth buffer, and the normal buffer of consecutive depth layers.

Edge Classification for CSG Models

Set operations applied to CSG primitives for composing CSG models produce surfaces whose edges generally do not correspond to the edges of the original polygonal meshes. As already announced in Section 4.2, image-based CSG algorithms clip part of the mesh without adding new polygons. It can be seen that visually important edges of the resulting CSG model do not necessarily correspond to the polygonal edges of its CSG primitives' meshes. Figure 5-11a illustrates the visually important edges of a CSG model (yellow) that are, in particular, independent of the edges of its underlying meshes (black).

The definition of visually important edges needs to be altered slightly to match the terms used for CSG models. Hence *silhouette edges* of CSG models represent edges where part of a front surface joins part of a back surface. *Crease edges* represent edges where two front surfaces or two back surfaces of CSG primitives join and form a certain angle. Since CSG models are solid objects, border edges do not exist at all. Figure 5-11b illustrates the silhouette and crease edges produced by set operations.

Design and Spatial Layout of CSG Models

Once generated, one can compose order-independent transparent depictions of CSG models [64] using the color buffers captured as color layers maps and the depth layers maps for depth sprite rendering (see Figure 5-12a). Although order-independent transparency allows one to illustrate the outer and inner faces of CSG models, and therefore represents a novel depiction technique for CSG, it is not sufficient for visualizing them adequately, because the outlines are hardly visible.

Based on the normal buffer available as the color layer map and the depth buffer available as the depth layer map, the edge map can be constructed for each depth layer. In this way, both

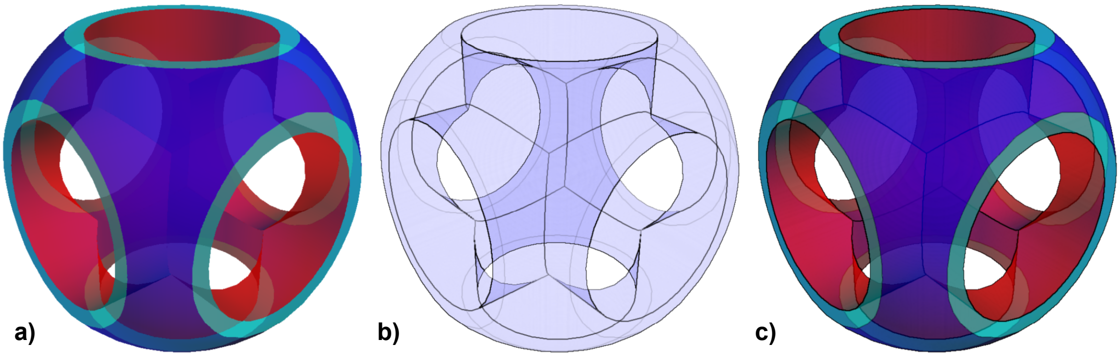


Figure 5-12: **Visualizations of CSG Models.** (a) Order-independent transparency rendering of CSG models shows their outer and inner faces, (c) a blueprint rendering enhances the outer and inner outlines and provides depth complexity cues, and (c) edge-enhanced order-independent transparency rendering visualizes the design as well as the spatial assembly of CSG models efficiently.

blueprints that outline a CSG model's outer and inner features and provide depth complexity cueing for it (see Figure 5-12b), and edge-enhanced order-independent transparent depictions that illustrate a CSG model's design and spatial layout efficiently (see Figure 5-12c) can be generated.

Performance Considerations

The programs for illustrating the design and spatial layout of CSG models using blueprint rendering run at interactive frame rates on today's graphics cards.

A fundamental graphics operation is depth peeling to process all the depth layers of a CSG model up to its depth complexity. As already mentioned in Section 5.2, the first few layers are generally sufficient to visualize 3D models efficiently. This can be observed for the CSG model in Figure 5-13. Here, two (a), three (b), and four (c) depth layers are used to depict the spanner. The differences in the visual quality in Figure 5-13b and Figure 5-13c are hardly noticeable. However, the depth complexity of the spanner was even higher. Restricting the number of rendering passes can thus increase performance without a significant loss of visual quality.

The performance of the blueprint rendering technique for CSG models is essentially bound to (1) the depth complexity or the number of rendering passes, (2) the window resolution, (3) the number of CSG primitives and their geometric complexity, and (4) the layout of the normalized CSG tree, or the set operations that are used.

The widget in Figure 5-10a takes 38.6 fps for the transparency rendering (see Figure 5-12a), 30.4 fps for the blueprint depiction (see Figure 5-12b), and 30.1 fps for the edge-enhanced transparency rendering (see Figure 5-12c) while considering 4 depth layers at a window resolution of 512×512 on a GeForce 6800GT graphics card. The spanner in Figure 5-13d takes 9.2 fps for the transparency rendering (see Figure 5-13e), 8.6 fps for the blueprint depiction (see Figure 5-13f), and 8.1 fps for the edge-enhanced transparency rendering using the same test conditions (see Figure 5-13g). It is noticeable that the spanner takes 8.7, 7.1, and 7.0 fps for rendering at a window resolution of 1024×1024, that is, the rendering performance decreases only slightly with higher window resolution.

5.7 Conclusions

This chapter has presented an image-space rendering technique that allows one to accentuate a 3D model's visible and occluded edges; it is based on the edge-map concept. The resulting

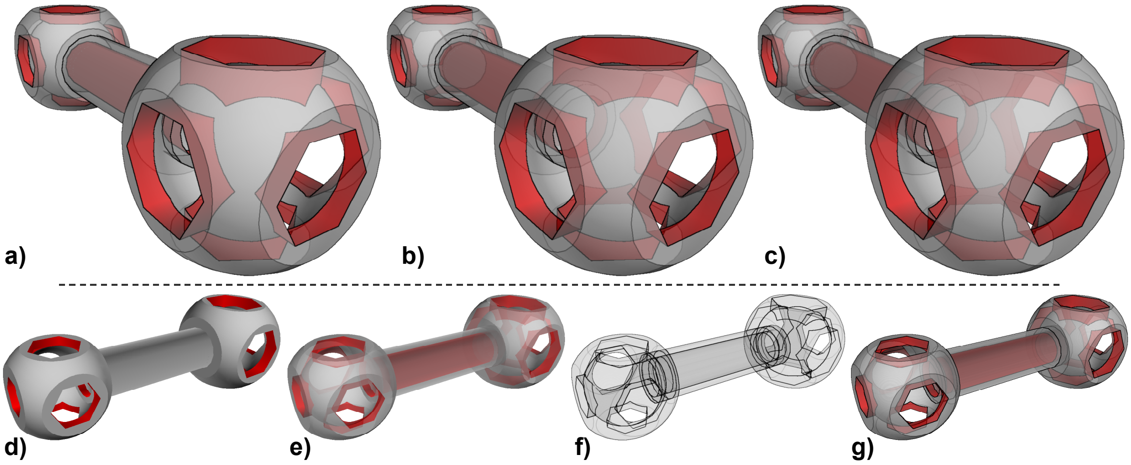


Figure 5-13: **Illustrating the Design of a Spanner.** Edge enhanced transparency renderings of the CSG model using two depth layers (a), three depth layers (b), and four depth layers (c). The spanner rendered with an image-space CSG rendering technique (d), the transparency rendering technique (e), the blueprint rendering technique (f), and the edge-enhanced transparency rendering technique (g).

blueprints provide comprehensible insights into complex aggregate object assemblies, such as mechanical parts and architecture.

It has been observed that the first few depth layers contribute to a comprehensible depiction whereas the remaining depth layers have less visual impact. Dynamically adjusting the number of rendering passes subject to the number of contributing pixels can thus increase performance while maintaining the visual quality of the blueprints.

It has also been observed that models of high structural complexity become increasingly difficult to perceive. Orthographic views can reduce the visual complexity in blueprints while still providing a comprehensible insight into the models, because most edges are parallel and superimpose on one another. Plan views of architecture, for instance, allow one to identify each of the building's components even when the structure is complex (see Figure 5-8). However, perspective views communicate spatial relationships more expediently. Depth masking can reduce the visual complexity in these views as well. Here, the model's front-most parts up to a highlighted feature can be clearly identified. However, the model's rear parts are neglected by depth peeling and thus cannot be perceived at all (see Figure 5-7). Blueprint rendering thus requires a more general method for reducing the visual complexity of the resulting depictions if the 3D model's structural complexity is more than can be reasonably displayed. Viewers could then perceive even the rear parts efficiently.

A disadvantage of blueprint rendering results from its image-space nature: although a 3D model's interior composition can be illustrated up to its depth complexity, additional visual cues such as dashed or dotted line styles for occluded edges are missing. In contrast to object-space algorithms (see Figure 2-4), individual line styles that allow one to perceive occluded edges cannot be applied to image-space edges. Since line styles are required to increase the visual perception of spatial arrangements, they should be addressed by future work. One could, for instance, first operate on edge maps by way of a *dilation filter* [85] to customize the line widths of consecutive depth layers. Perhaps dotted or dashed line styles could be produced in a similar way.

The blueprint rendering technique maintains a correct depth behavior for depth testing when blending the depth layers' results into the frame buffer. Thus, 3D models rendered as blueprints can be combined with arbitrary 3D scene contents. As with transparency rendering, one must ensure that the scene's opaque models are rendered first, so that the scene's models declared for blueprint rendering can be added smoothly. In conclusion, blueprint rendering can be an effective tool in interactive applications, such as CAD or CAM systems.

Outlining spatial relations is particularly useful for conveying the composition of CSG models. So, an extension of blueprint rendering to image-space CSG algorithms, which synthesizes the blueprints and edge-enhanced transparent depictions, has been introduced in this chapter as well. These depictions can visualize the design and spatial assembly of CSG models in a way that assists their interactive construction.

Chapter 6

“Sketchy Drawings”

Rendering in a “sketchy” manner is of vital importance for communicating visual ideas and for illustrating the preliminary state of a draft or concept, especially in application areas such as archeology and product and architectural design [107].

Common photorealistic renditions are often less efficient in proposing ideas and concepts, because they imply the impression of finality and correctness. Photorealistic renderings thus reduce one’s ability to rethink enhancements and modifications. In contrast, sketches communicate visually and can therefore be more helpful when dealing with renditions. In particular, sketches encourage the exchange of ideas when people are reconsidering drafts; sketches express uncertainty and suggest work in progress. In fact, hand-drawn sketches are still an integral part of the development process in architectural or product design; in the film making process storyboards are still used to assist communication (see Chapter 7). Sketches allow one to present drafts that suggest the possible design of an object’s layout in cases when its precise composition is unknown, e.g., in archeology. As stated in Section 2.2, STROTHOTTE ET AL. argue in favor of uncertainty when presenting and communicating drafts of reconstructions of ancient and medieval architecture. They implement a sketch renderer for generating sketchy depictions of architecture to express “imprecision, incompleteness, and vagueness” [114].

Sketchy drawing is a real-time rendering technique for sketching visually important edges and inner color patches of arbitrary 3D geometries non-uniformly even beyond the original models’ geometric boundaries [89][93]. Generally speaking, with sketchy drawing one can sketch the 3D geometries’ outlines to imply vagueness and “crayon in” inner color patches extending beyond the sketchy outline as though they had been painted in roughly. Combining both techniques produces sketchy, cartoon-like depictions that can enhance the visual attractiveness of 3D sceneries’ images and, more importantly, can increase one’s ability to review drafts and encourage discussions.

In order to express vagueness, sketchy drawing implements uncertainty based on Perlin noise [99] and it applies the resulting uncertainty values in image-space for sketching. When interacting with the 3D scene this approach allows one to maintain *frame-to-frame coherence*, that is, to preserve form and style in images of consecutive frames and to avoid undesirable “flickering”. Alas, the approach results in the *shower-door-effect*, that is, the resulting sketchy depictions appear to “swim” in image space as if viewed through ripped glass. In order to reduce the shower-door-effect, the sketchy drawing rendering technique determines the uncertainty values that have an obvious correspondence to the geometric properties of the 3D geometry [91].

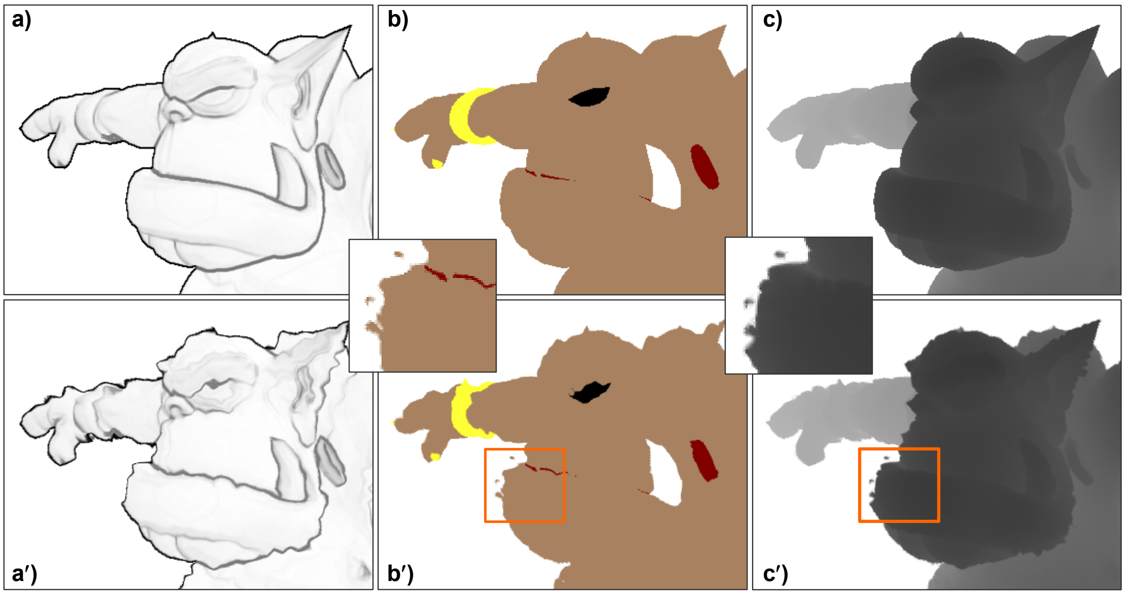


Figure 6-1: **Applying Uncertainty to Edge and Shade Maps.** (a) The edge map and (b) shade map are two ingredients for sketchy drawing. Applying uncertainty results in (a′) perturbations of the edge map and (b′) perturbations of the shade map. Depth sprites use the (c) depth map to adjust z-values. For sketchy rendering, the uncertainty applied to the edge and shade map are also applied to perturb the depth map (c′) for depth sprite rendering.

Since a blueprint represents a special form of a draft used for representing complex assemblies, both sketchy drawing and blueprint rendering (see Chapter 5) are combined in order to depict the assemblies’ composition in a sketchy way.

The remainder of this chapter is structured as follows: Section 6.1 introduces the ingredients for sketching and Section 6.2 gives a brief overview of Perlin noise. The sketchy drawing rendering technique is divided into two parts: Section 6.3 presents how uncertainty values are applied to synthesize sketchy depictions and Section 6.4 presents how depth information is determined in order to combine sketchy depictions with 3D scenery. Section 6.5 outlines applications and Section 6.6 presents further style variations. Section 6.7 presents an approach for controlling uncertainty in order to reduce the shower door effect. Section 6.8 presents the combination of both sketchy drawing and blueprint rendering that allows one to sketch the assembly of complex aggregate 3D models. Section 6.9 draws final conclusions.

6.1 Edges and Color Patches

Edges as well as solid color patches derived from 3D models are considered as ingredients for implementing the sketchy drawing rendering technique. Both are rendered in a sketchy way. For this, they are preserved as intermediate rendering results using textures. The edge-enhancement technique (see Chapter 4) computes the 3D models’ visually important edges as an edge map (see Figure 6-1a). For edge-map construction, the z-buffer’s contents are captured as high-precision texture, the depth map (see Figure 6-1c, Sec. 4.7), because the sketchy drawing rendering technique later on requires the 3D models’ z-values. Furthermore, the surface colors of unlit 3D models are rendered to texture, producing solid color patches that appear flat, cover all surface details, and emulate a cartoon-like style. This texture is referred to as the *shade map* (see Figure 6-1b).

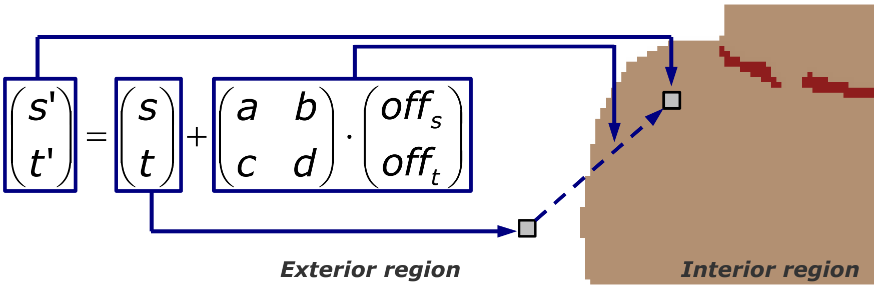

$\begin{pmatrix} s' \\ t' \end{pmatrix} = \begin{pmatrix} s \\ t \end{pmatrix} + \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} off_s \\ off_t \end{pmatrix}$
Exterior region
Interior region

Figure 6-2: **Applying Uncertainty in Image Space.** The product of the uncertainty value (off_s,off_t) derived from the noise texture and a 2×2 matrix (with weights a, b, c, and d) forms the degree of uncertainty that is applied to the texture coordinates (s,t) of a fragment to translate them in image space. Here, the perturbed texture coordinates (s',t') access a texture value of the shape map's interior region, even though its initial texture coordinates (s,t) would access the exterior region.

6.2 Perlin Noise

We look for uncertainty values that let one sketch edges and color patches pseudo-randomly in image space. PERLIN introduces a pseudo-random noise function that serves as a primitive for controllable noise [99]. The stochastic patterns produced by the *Perlin noise function* maintain spatial coherence, that is, noise values calculated for "adjacent" input parameters are correlated to one another. Perlin noise allows one to simulate a wide variety of natural phenomena. One application of Perlin noise is the implementation of procedural texturing, e.g., for creating the impression of natural-looking materials, without the necessity to generate an explicit image texture. The *turbulence function* combines several calls to the Perlin noise function implementing a stochastic function. EBERT ET AL. modify texture coordinates in image space using a stochastic function to perturb image textures [33]. Sketchy drawing applies Perlin noise in a similar way in order to generate sketchy depictions that maintain frame-to-frame coherence through interaction and animation.

6.3 Applying Uncertainty

The sketchy drawing technique applies uncertainty values to edges and surface colors in image space in order to simulate the effect of "sketching on a flat surface." For that purpose, the technique textures a screen-aligned quad (filling in the viewport of the canvas) using the edge and shade maps as input. Moreover, the technique applies an additional texture, whose texture values represent uncertainty values. Because we want to achieve continuous sketchy boundaries and frame-to-frame coherence, sketchy drawing applies a noise texture whose texture values have been determined by the Perlin noise function; hence neighboring uncertainty values are correlated in image space. Once created (in a preprocessing step), the noise texture serves as an offset texture for accessing the edge and shade maps when rendering. That is, its texture values slightly perturb the texture coordinates of each fragment of the quad that accesses the edge and shade maps; see Section 3.5.

In addition, sketchy drawing introduces a *degree of uncertainty* to control the amount of perturbation, for which a user-defined 2×2 matrix is used. The rendering technique multiplies uncertainty values derived from the noise texture by that matrix to weight all these values uniformly and then uses the resulting offset vector to translate the texture coordinates. Figure 6-2 illustrates the perturbation of the texture coordinates that access the shade map using the degree of uncertainty.

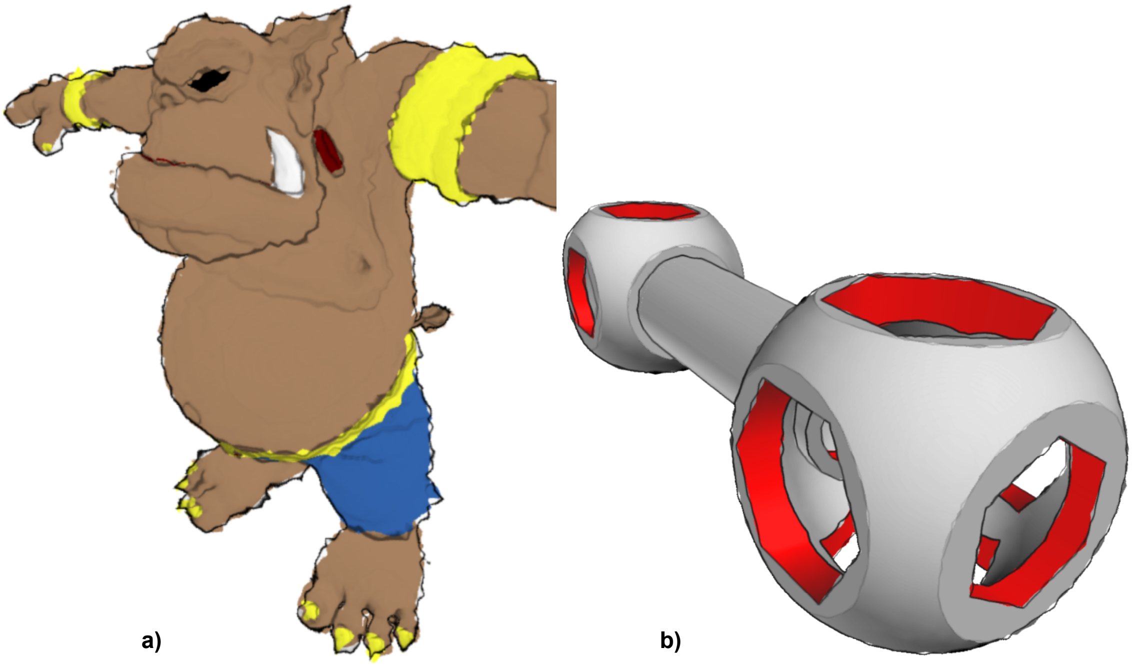


Figure 6-3: **Sketchy Drawings for Sketching 3D Models.** (a) The sketchy depiction of the Ogre shows spots near his mouth. (b) In contrast to the original approach to sketchy drawing one can also use lit surfaces to generate the shade map, e.g., for sketching CSG models.

In order to enhance the sketchiness effect, sketchy drawing perturbs the texture coordinates for accessing the edge and shade maps differently. It thus applies two different 2×2 matrices, resulting in different degrees of uncertainty for each map. One degree of uncertainty shifts the texture coordinates of the edge map, and one shifts the texture coordinates of the shade map, that is, sketchy drawing shifts them in contrary directions. Figure 6-1a′ shows the edge map and Figure 6-1b′ shows the shade map after uncertainty has been applied.

We denote the set of texels that correspond to fragments of 3D geometry as *interior regions*; the set of texels that do not correspond to fragments of 3D geometry is called *exterior regions*. So, in conclusion, by texturing the quad and perturbing texture coordinates using uncertainty, the sketchy drawing technique can access interior regions of the edge and shade maps, although the initial texture coordinates would access exterior regions and vice versa (see Figure 6-2). In this way, interior regions can be sketched beyond the 3D geometry's boundary, and exterior regions can penetrate interior regions. Sketchy drawing can even produce spots beyond the geometric boundary (see Figure 6-1).

Sketchy drawing finally combines the resulting texture values of both the edge and the shade maps. For this, the intensity values derived from perturbing the edge map is multiplied by the color values derived from perturbing the shade map. For the sketchy depictions in Figure 6-3, uncertainty values (off_s, off_t) are determined using the turbulence function [33]:

```
off_s ← turbulence(s,t); // Calculating uncertainty based on s and t
off_t ← turbulence(1-s,1-t);         // and based on (1-s) and (1-t)
```

6.4 Adjusting Depth

Up to now, just a screen-aligned quad is textured with a depiction synthesized by the sketchy drawing rendering technique. This method has the following significant shortcomings: when

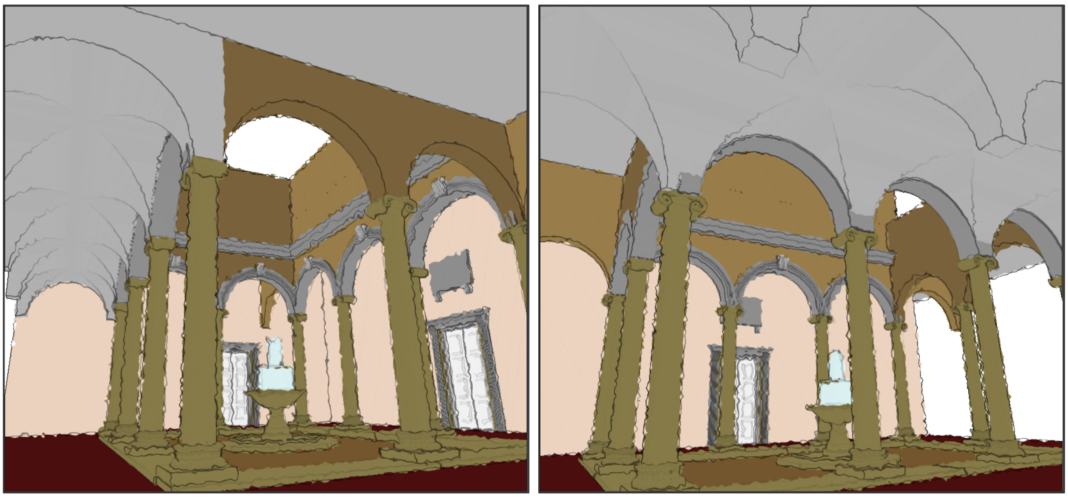


Figure 6-4: **Drafts of Medieval Architecture.** Sketchy depictions can communicate uncertain knowledge to present, for instance, cloister's possible reconstruction.

rendering a quad that is textured with textures derived from 3D geometry, (1) z-values of the original geometry are not present in interior regions, and in particular (2) no z-values of the original 3D geometry are present in exterior regions when uncertainty has been applied. In conclusion, a sketchy depiction cannot interact correctly with other objects in the 3D scene.

To overcome these shortcomings, sketchy drawing applies depth sprite rendering while considering the previous perturbations. That is, sketchy drawing additionally textures the quad with the high-precision depth texture, which is already available as a depth map (see Figure 6-1c), and accesses this texture twice using perturbed texture coordinates. As first perturbation, sketchy drawing applies the degree of uncertainty used for accessing the edge map; as second perturbation, the technique applies the degree of uncertainty used for accessing the shade map. The minimum value of both these texture values produces the final z-value of the fragment for depth testing. Figure 6-1c′ shows the result of both perturbations applied to the depth map. The interior region of the perturbed depth map matches the combination of the interior regions of both the perturbed edge map and the perturbed shade map. Even those spots produced for the shade map appear in the perturbation of the depth map (see Figure 6-1).

Modifying depth sprite rendering thus allows one to adjust the z-values of a screen-aligned quad that is textured by the 3D model's sketchy depiction. In this way, sketchy drawing behaves correctly with respect to depth, that is, the z-buffer remains in a correct state with respect to the geometry of that model. The 3D model's sketchy representation can thus be combined with further (e.g., non-sketchy) 3D models.

The sketchy drawing rendering technique facilitates real-time frame rates. The Ogre in Figure 6-3, for instance, takes 49.0 fps at a window resolution of 1024×1024. Listing 6-1 illustrates the fragment shader for implementing the sketchy drawing rendering technique.

Listing 6-1. **Fragment Shader for Implementing Sketchy Drawings.** The fragment shader perturbs texture accesses for implementing sketchy drawings.

```
uniform sampler2D edgeMap;      // 2D texture with edge intensities
uniform sampler2D shadeMap;     // 2D texture with color values
uniform sampler2D depthMap;     // 2D texture with depth values
uniform sampler2D noiseTexture; // 2D texture with offset values
uniform vec2 windowDimension;
uniform mat2 matEdges;
uniform mat2 matShades;

void main (void) {
  // Calc. texture coordinates that corresond to pixel position
  vec4 texCoord = vec4(gl_FragCoord.x/windowDimension.x,
                       gl_FragCoord.y/windowDimension.y, 1.0, 1.0);
  texCoord = gl_TextureMatrix[0]*texCoord;

  // Access noise texture
  vec2 noiseValue = texture2D(noiseTexture, texCoord.xy).xy;

  // Calc. texure coordinate for accessing the edge map
  vec2 edgeTexCoord = texCoord.xy + (matEdges*noiseValue);

  // Calc. texure coordinate for accessing the shade map
  vec2 shadeTexCoord = texCoord.xy + (matShades*noiseValue);

  // Access depth map twice using perturbations of edges and shades
  // and set the new z-value to the minimum of both
  float edgeDepth = float(texture2D(depthMap, edgeTexCoord).xyz);
  float shadeDepth = float(texture2D(depthMap, shadeTexCoord).xyz);
  float depth = min(edgeDepth, shadeDepth);

  // Either discard fragment in advance or proceed with
  // new z-value for the depth sprite rendering
  if(depth == 1.0) {
    discard;
  }
  gl_FragDepth = depth;  // New z-value for depth testing

  // Access edge map
  vec4 edgeValue = texture2D(edgeMap, edgeOffset);
  if(edgeDepth == 1.0) {
    edgeValue = vec3(1.0,1.0,1.0,1.0);  // white background
  }

  // Access shade map
  vec4 shadeValue = texture2D(shadeMap, shadeOffset);
  if(shadeDepth == 1.0) {
    shadeValue = vec3(1.0,1.0,1.0,1.0);  // white background
  }

  // Combine both color values
  gl_FragColor = edgeValue*shadeValue;
}
```

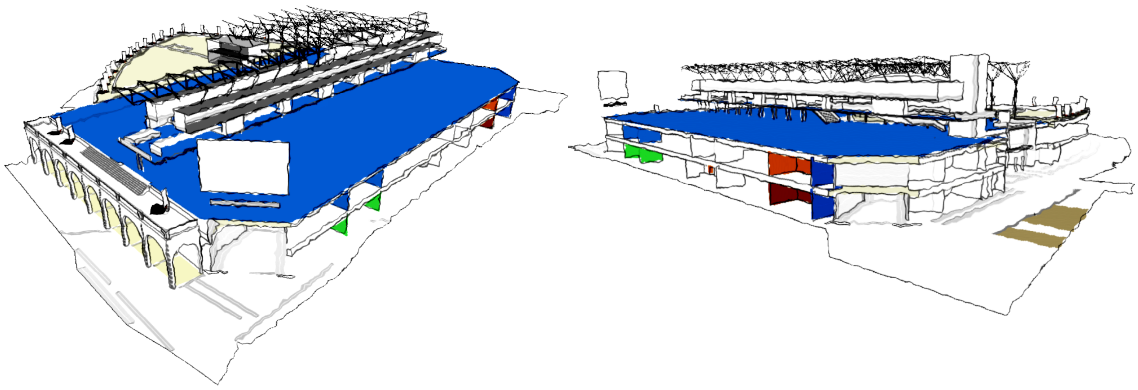


Figure 6-5: **Drafts of Modern Architecture.** Sketchy drawings can illustrate design decisions to build, for instance, a department store.

6.5 Drafts of Architecture

When knowledge about the precise composition of architecture is not available or is partially missing, which is often the case for ancient or medieval buildings, we can generate a sketchy depiction of that architecture to help visualize a possible reconstruction. The imprecision conveyed by the sketches lets viewers understand that the presented draft merely depicts one reconstruction among many possible alternatives (Sec. 2.2). Figure 6-4 shows sketchy depictions of architecture presenting a virtual reconstruction of a cloister.

Furthermore, sketchy drawing can generate architectural drafts that illustrate the design of modern architecture. In this way, a preliminary state or design decisions for the buildings can be presented in a noncommittal way. Sketchy representations of architecture can then help when reconsidering the design in order to convince customers of the soundness of the underlying concept or to encourage their participation in the design and the discussion of further decisions (Sec. 2.2). Figure 6-5 shows sketchy depictions of the designs for a department store.

6.6 Variations of Sketchy Drawings

This section presents two variations of the sketchy drawing rendering technique, both of which are certainly real-time capable as well.

Roughened Profiles and Color Transitions

Although edges and surface colors are sketched non-uniformly, the profiles and the color transitions of a sketchy depiction look exactly as though they had been sketched with pencils on a flat surface. Sketchy drawing can roughen the profiles and color transitions to simulate different drawing tools and media, such as chalk, applied on a rough surface. For this, random noise values are applied; hence, adjacent texture values of the noise texture are uncorrelated. Consequently, the degrees of uncertainty that perturb the texture coordinates of adjacent fragments are also uncorrelated. In this way, sketchy drawing can produce depictions with softened and frayed edges and color transitions (see Figure 6-6a). The roughness and granularity, in particular for edges, vary as though the pressure had varied as it does when drawing with chalk. This effect depends on the amount of uncertainty applied in image space.

Repeated Edges

A fundamental technique in hand drawings is to repeatedly draw edges to draft a layout or design. Sketchy drawing can restrict sketchiness to visually important edges only in order to

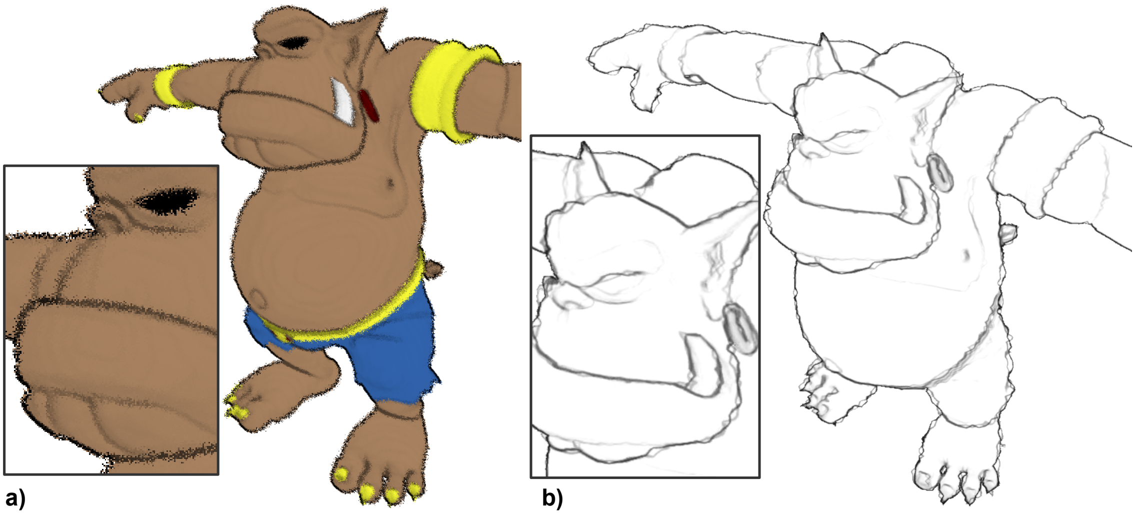


Figure 6-6: **Different Styles of Sketchy Rendering.** Sketchy drawings enable one to simulate (b) roughened profiles and color transitions and (c) repeated edges.

simulate this technique. For that purpose, sketchy drawing excludes the shade map but applies the edge map multiple times using different degrees of uncertainty and possibly different edge colors. Edges will then overlap non-uniformly as if the edges of the 3D model had been sketched repeatedly (see Figure 6-6b). Certainly sketchy drawing must also adjust depth information by accessing the depth map multiple times using the corresponding degrees of uncertainty.

6.7 Controlling Uncertainty

Controlling uncertainty values, in general, enables one to configure the visual appearance of sketchy depictions. By providing uncertainty values based on a Perlin noise function for each pixel in image space, (1) we can access interior regions from beyond exterior regions and vice versa to sketch beyond the boundary of 3D geometries, and (2) we achieve frame-to-frame coherence for sketchy drawing, for instance, when interacting with the 3D scene (because neighboring uncertainty values are correlated). However, uncertainty values remain unchanged in image space and have no obvious correspondence to the geometric properties of the original 3D geometry. Consequently, the resulting sketchy depictions tend to "swim" in image space, known as the shower-door effect, and a sketchy depiction's appearance cannot be predetermined, which is essential when considering artistic guidelines.

To overcome these limitations, sketchy drawing must accomplish at least the following:

- Preserve geometric properties, such as surface positions, normal vectors, or curvature information, for determining uncertainty values.
- Continue to provide uncertainty values in exterior regions, at least close to the 3D geometry.

Preserving Geometric Properties

To preserve the geometric properties of 3D geometry in order to control uncertainty, the sketchy drawing rendering technique proceeds as follows:

1. It renders the geometric properties directly into a texture to generate an additional G-Buffer.

2. Next it textures the screen-aligned quad with that additional texture, and then accesses geometric properties using texture coordinates (*s*,*t*).
3. Finally, it calculates uncertainty values based on a noise function, using the geometric properties as parameters.

Sketchy drawing can then use these uncertainty values to determine the different degrees of uncertainty needed to perturb the texture coordinates resulting in (s', t'). From a mathematical point of view, the rendering technique uses the following function to determine the perturbed texture coordinates (s', t'):

$$f : (s, t) \rightarrow (s', t')$$
$$f(s, t) = p\big(s, t, g(s, t)\big),$$

where (*s*,*t*) represent a fragment's texture coordinates produced when rasterizing the screen-aligned quad, *g*() provides the geometric properties available in the additional texture, and *p*() determines the perturbation applied to (*s*,*t*) using *g*() as input.

Sketchy drawing requires two different functions *f*(*s,t*) in order to handle perturbations of the edge map ($f_{Edge}(s,t)$) and the shade map ($f_{Shade}(s,t)$) differently.

Enlarging the Geometry

The sketchy drawing technique enlarges the original 3D geometry to generate the geometric properties in its surroundings in image space. Sketchy drawing implements this by shifting the vertex of its 3D meshes slightly along its vertex normal in object space. For this technique to work as expected, the surface must at least form a connected component and each of its shared vertices must provide an interpolated normal vector.

Enlarging the 3D geometry in this way allows sketchy drawing to render the geometric properties into a texture for calculating uncertainty values in interior regions as well as in the exterior regions (near the original 3D geometry in image space). Thus, uncertainty values can be determined and applied so that interior regions can be sketched beyond the 3D geometry's boundary and exterior regions can penetrate interior regions. In conclusion, sketchy drawing can apply perturbations based on uncertainty values that do have an obvious correspondence to the underlying 3D geometry.

Reducing the Shower-Door Effect

The following example demonstrates how to control the sketchiness in order to reduce the shower-door effect.

The sketchy drawing rendering technique renders enlarged 3D geometry with its object-space positions as color values into a texture. To do so, the technique determines the object-space position for each displaced vertex and provided them as texture coordinates to the rasterization process. The rasterizer then produces interpolated object-space positions for each fragment. A fragment shader then outputs them as high-precision color values to populate a high-precision texture (Sec. 3.3). In this way, *g*(*s*,*t*) preserves object-space positions.

Based on *g*(*s*,*t*), sketchy drawing can determine texture coordinates *f*(*s*,*t*) using *p*(). In this example, the function *p*() calculates the perturbation by a user-defined 2×2 matrix and by a Perlin noise function encoded into a 3D texture. The 3D texture can be accessed using g(*s*,*t*) as texture coordinates. Multiplying the resulting texture value by the 2×2 matrix gives the degree of uncertainty. The function *f*(*s*,*t*) finally applies the degree of uncertainty to perturb (*s*,*t*), resulting in (s', t').

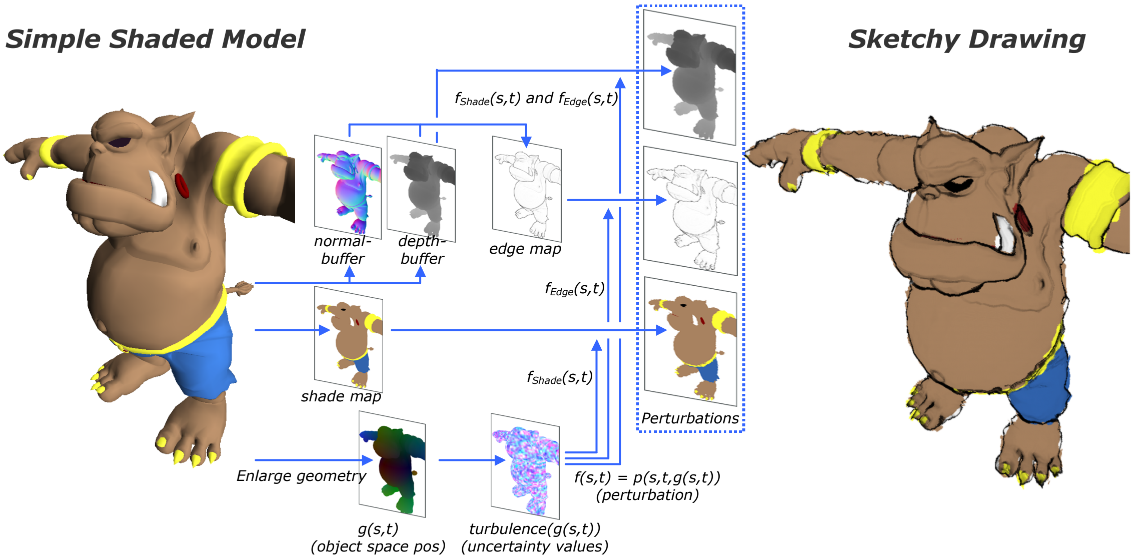


Figure 6-7: **Conceptual Sketch for Controlling Uncertainty.** The overview illustrates intermediate rendering results involved in the process for generating sketchy drawings. It clarifies the usage of *f*(*s*,*t*) when considering geometrical properties.

Calculating $f_{Edge}(s,t)$ and $f_{Shade}(s,t)$ using different matrices results in the sketchy depiction shown in Figure 6-7(right). The overview in Figure 6-7 illustrates the sketchy drawing's process flow for considering geometric properties.

Conclusions

Controlling uncertainty values reduces the shower-door effect. Moreover, the previous example gives a clue as to how to control sketchy depictions using geometric properties. However, further research is required to implement sketchy drawing that is capable of simulating artistic styles.

A disadvantage of the technique presented here is that the enlarged geometry produces abrupt changes of geometric properties in image space at those locations where the values of the z-buffer change abruptly. This can result in disturbing artifacts when interacting with the 3D scene, because neighboring uncertainty values here are not correlated to one another any more.

6.8 Sketching Blueprints

Since blueprint rendering provides spatial insight into aggregated objects for understanding them as a whole (see Chapter 5) and sketchy drawing communicates drafts for the purpose of reconsideration, the combination of both rendering techniques is the obvious choice for illustrating the design decisions of complex aggregate objects.

Combining sketchy drawing and blueprint rendering is a straightforward task: as before 3D geometries are graphically decomposed into disjunctive layers to generate color layer maps, which represent the color buffer and the normal buffer of each depth layer, and depth layer maps. As variation to the original sketchy drawing rendering technique, lighting calculations are used to determine the color values that populate the color buffer; the color layer map then represents the shade map for each layer (see Figure 6-3b). In this way, spatial cues are provided (Sec. 5.3). The edge map can then be constructed for each depth layer (see Figure 6-8a). In contrast to the original blueprint rendering technique, the noise texture is applied to perturb the texture coordinates for final image composition. That is, a sketchy depiction is rendered for each

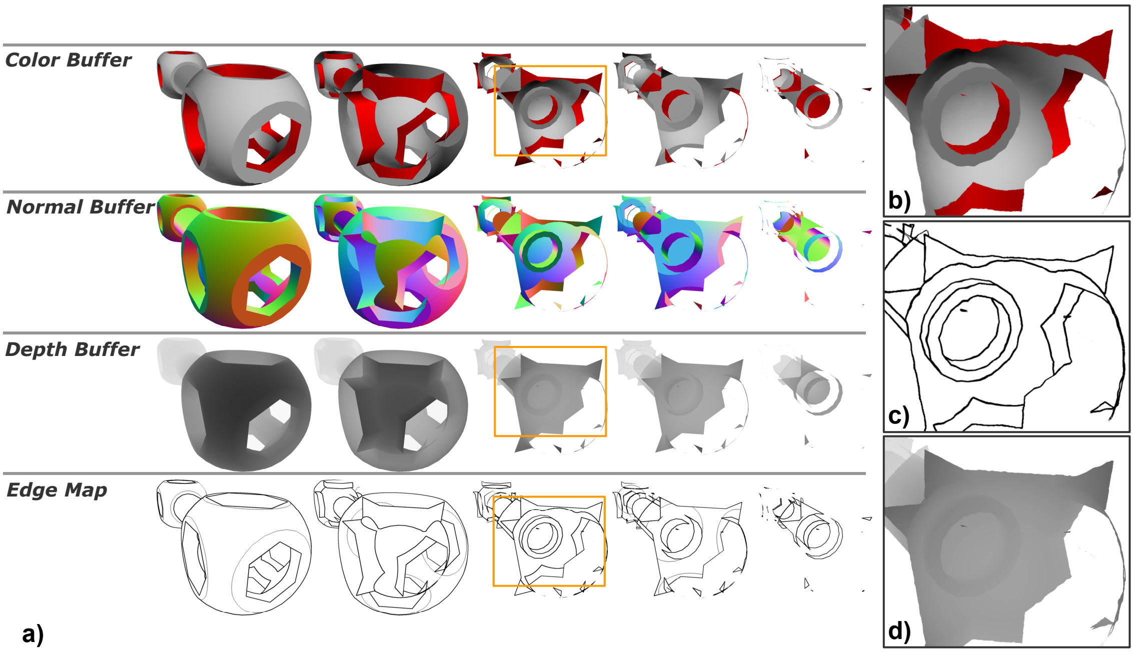


Figure 6-8: **Layer Maps for Composing Sketchy Blueprints.** Depth layer maps and color layer maps are generated and the edge maps are then constructed for each depth layer (a). Perturbing the contents of (b) the shade map, (c) the edge map, and (d) the depth map for each depth layer enables one to render layouts in a sketchy manner.

depth layer by combining the perturbation of the shade map (see Figure 6-8b) and the perturbation of the edge map (see Figure 6-8c). The resulting sketchy depictions are then blended in depth-sorted order into the frame buffer using the perturbed depth layer maps (see Figure 6-8d) for depth sprite rendering.

As long as the rendering technique uses the same noise texture for all textures derived from all the depth layers, consecutive layers can still be blended into the frame buffer without disturbing artifacts, such as diverging edges or interpenetrating layers. Consider one and the same texel location of the textures of consecutive layers. The corresponding texture value is accessed by specific texture coordinates (s,t) sampling that location in all textures regardless of whether the texture coordinates result from a previous perturbation or not. Figure 6-9a illustrates that the edges of edge maps of consecutive depth layers match one another even though they have been perturbed. Consequently, the ordering of z-values specified by the ordering of depth layers can even be ensured for composing all sketchy depictions.

As one application, the design decisions for composing CSG models can be communicated by rendering their complex aggregation in a sketchy manner (see Figure 6-9b).

6.9 Conclusions

The sketchy drawing rendering technique presented here allows one to sketch both edges and color patches, an approach which has rarely been addressed by previous work. Furthermore, sketchy drawing is, by its image-space nature, both almost independent of the 3D model's geometric complexity, and essentially independent of the number of edges that are sketched. In addition, it requires few prerequisites from 3D geometry, e.g., per-vertex normal vectors. Sketchy drawing can avoid disturbing flickering effects when interactively exploring 3D models by ensuring frame-to-frame coherence. Furthermore, an approach for reducing the shower-door effect has been introduced.

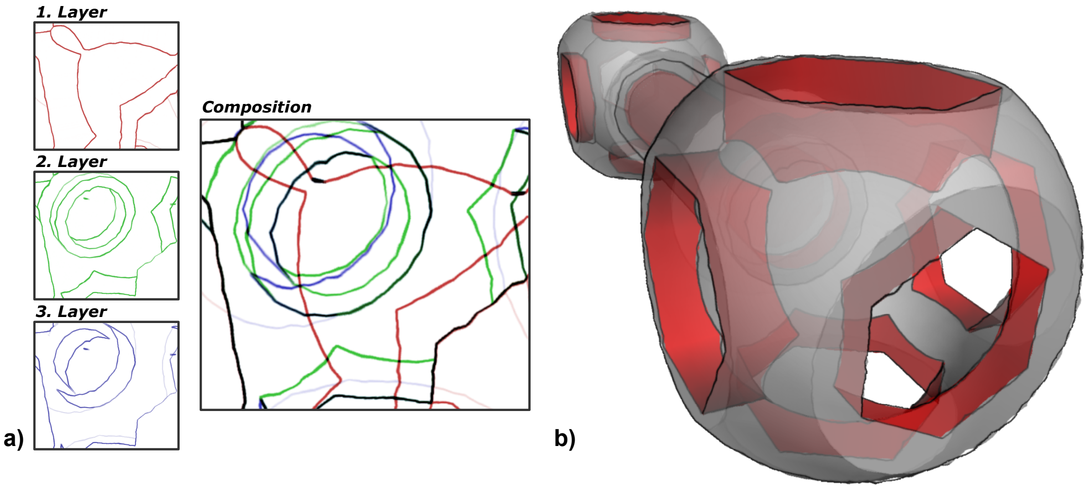


Figure 6-9: **Composing Sketchy Layouts.** (a) Repeated edges in edge maps of consecutive depth layer match to one another even though uncertainty has been applied. (b) Composing sketchy depictions of each depth layer results in the final sketchy depiction of the CSG model's layout.

Although an approach for controlling uncertainty has been presented, it does, in its current form, not allow one to control the artistic stylization of image-space edges. In contrast, object-space silhouette algorithms allow graphic designers to configure the artistic styles of individual line segments (Sec. 2.1) and allow one to maintain frame-to-frame coherence in interactive environments [59] as well. So, the stylization produced by sketchy drawing is visually inferior to that produced by object-space algorithms. One reason is that visually important edges stored in an edge map represent a composition of connected line segments in image space. Thus, visually important edges cannot be decomposed into individual line segments. Furthermore, edges are represented by pixels in image space, that is, they lack an analytic representation. Generating stroke-like geometry and applying artistic stylization using texture mapping can thus not be achieved by sketchy drawing. Today, computer graphics hardware supports texture lookups at the vertex processing stage and upcoming graphics hardware will support the generation of additional geometry while processing the rendering pipeline. Future implementations of sketchy drawing may thus be able to decompose the intensity values of the edge map into line segments for building brush-like strokes.

Nevertheless, sketchy drawing represents a real-time capable rendering techniques that allow one to stylize visually important edges using uncertainty. Sketching beyond the model's original geometric boundary while ensuring a correct depth behavior is managed by depth sprites that adjust depth information. Hence, a 3D model's sketchy depictions can be combined arbitrarily with further 3D scenery. In addition, the sketchy drawing rendering technique is implemented on a per-object basis and integrates into the programmable rendering pipeline as well. Sketchy drawing can thus be integrated into any real-time graphics application such as CAD or storyboarding systems.

Though accentuating visible and occluded edges in image space using blueprint rendering already represents a novel technique, incorporating sketchy drawing provides a new tool for interactive communication. Consequently, the design and layout of complex, aggregate assemblies, which can either be represented by polygonal 3D models or CSG models, can be sketched to encourage participation, discussions, and reconsideration.

Chapter 7

Depicting Dynamics

In visual art, a single static image frequently represents much more than projected 3D scenery. Artists are accustomed to include subtle visual elements outlining movements, indicating past or future events, sketching ongoing activities, or guiding the observer's attention. Artists have found ways to visualize the physical as well as the non-physical dynamics of scenes using graphics techniques. In a sense, we can consider these depictions as a form of expressive visual contents adopting the styles of visual art and abstraction techniques. These depictions can serve, for example, as pictograms and signs that advise and assist people or for creating effective advertising [86]; they are also omnipresent in comic books and storyboards that present dynamics and narrate sequential processes effectively (see Figure 7-1).

Taking these depictions a step further, we can speak of *smart depictions*, that is, depictions that "capitalize on the humans' facility for processing visual information and thereby improve comprehension, memory, inference, and decision making" [1]. The developed depiction system [88] presented in this thesis automatically generates smart, compelling depictions of dynamics from a general 3D scene description. The dynamics are depicted following the traditional design principles of visual art and visual narrations, and the principles of classic graphics design such as those found in comic books [81] and storyboards [8][60]. These media offer a rich vocabulary of visual art deployed as techniques to facilitate the visual communication of a wealth of activities and events in static images. In particular, we can symbolize in a single, static image past, ongoing, and future activities as well as events taking place in 3D scenes. Additionally, the system takes into account non-visual information. For example, in the scope of narratives the system can integrate information such as tension, danger, and other emotions into the symbolization process.

Designers and artists traditionally depict dynamics by hand or utilize imaging or sketch tools. The challenge was to find a solution for automating the process of specifying, interpreting, and mapping the dynamics of visual narrations in 3D scenes. The method presented here uses the following approach:

- Depiction techniques analyze scene and behavior descriptions (e.g., encoded in scene graphs), and map found and relevant dynamics to dynamics glyphs.
- Dynamics glyphs are additional graphical elements that symbolize dynamics and augment the resulting image of the 3D scene.

Designers can configure the way depiction techniques operate, and they can edit the visual and textual appearance of dynamics glyphs. Consequently, the system lets one model "from word to image" [8].

As an enabling technology, the system uses non-photorealistic rendering intensely [44][116]. In fact, digitizing the process of generating visual art is increasingly feasible because of expressive and artistic rendering algorithms, most of them now operating in real-time, e.g., edge-

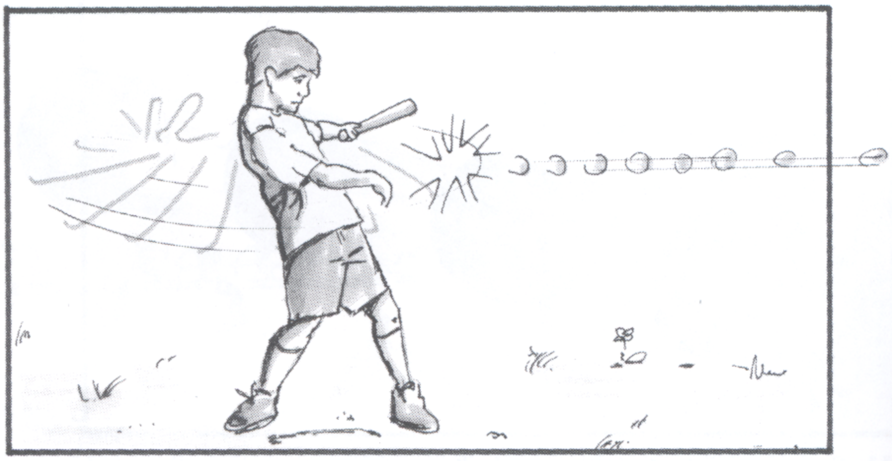


Figure 7-1: **Storyboard of a Batting Sequence.** A storyboard depiction illustrates the batting sequence of a boy playing baseball [60]. The dynamics depicted include the swing of the baseball bat, the bat hitting the ball, and the accelerated ball.

enhancement (see Chapter 4), blueprint rendering (see Chapter 5), and sketchy drawing (see Chapter 6).

Compared to imaging and sketch tools the dynamics-depiction system offers the following benefits:

- It explicitly models the graphical representations of dynamics.
- It applies to a standard 3D scene description, and it integrates smoothly into a scene graph based graphics system.
- It supports design alternatives by selecting specific types of dynamics and by configuring the symbolization process. The user can experiment and finally choose those depictions that best communicate her or his ideas.
- A user can easily modify a given depiction and accept and adopt changes after a reconsideration phase to start the next iteration step.

The manifold scenarios, in which smart depictions can be applied, include:

- Non-artists can model and generate smart depictions of dynamics and visual narration from common scene descriptions based on the system’s built-in analysis and symbolization capabilities.
- Graphics artists can customize and automate the production of smart depictions.
- In planning and discussion processes, smart depictions provide content-rich static imagery well-suited to be a basis for manual and cooperative sketching and illustrating.
- In the pre-production phase of motion-picture productions, smart depictions serve as storyboard-like depictions derived from a previsualization of the scene including its narration and intended dynamics.

This chapter is structured as follows: Section 7.1 provides a brief survey of the principles and guidelines for depicting dynamics in static images and outlines previous work. Section 7.2 then introduces dynamics glyphs that symbolize dynamics. Section 7.3 introduces scene and behavior graphs as general tools for scene and animation specification for deriving and interpreting dynamics. Section 7.4 provides a formal description of the process of assembling dynamics information in scene and behavior graphs. Section 7.5 gives a characterization of dynamics to type and access motion for symbolization. Section 7.6 presents depiction techniques that interpret dynamics and implement rendering techniques for generating dynamics

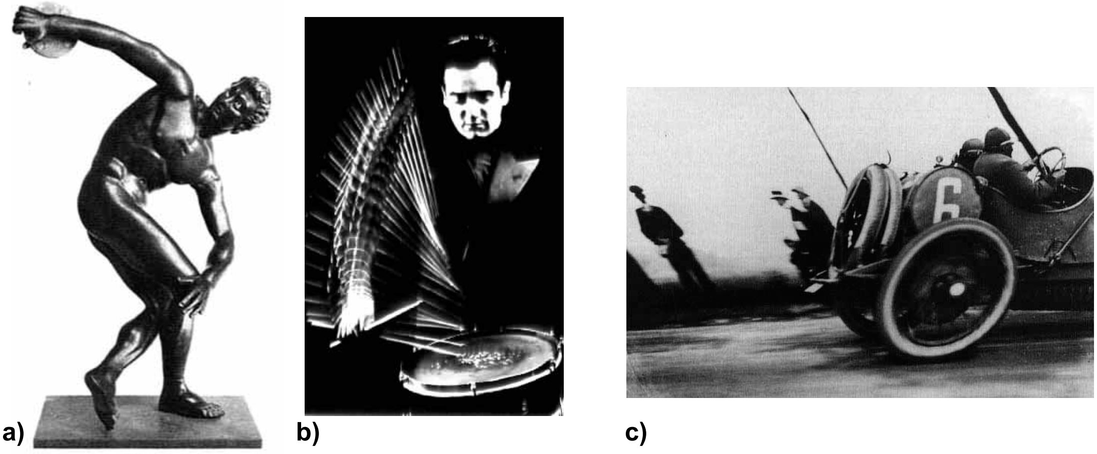


Figure 7-2: **Representations of Motion.** (a) Broken symmetry, (b) multiple stroboscopic images, (c) and forward lean show motion in static images.

glyphs. Section 7.7 introduces expressions that permit a high degree of user involvement to model and enhance the explanatory power of the resulting depictions. Section 7.8 then gives a overview on the entire depiction system and outlines user involvements. Section 0 introduces user-defined semantics and semantics-related depictions for narrating processes more vividly, e.g., frames illustrating a camera in motion or bubbles depicting a talking character. Section 7.10 finally focuses on pattern-based symbolizations and considers more complex relationships between scene and behavior graphs, automatic enhancements for expressions, and dynamics influence one another and Section 7.11 gives conclusions.

7.1 Introduction to Depictions of Dynamics

In the field of visualizing dynamics, CUTTING [18] surveys the traditional techniques for depicting motion in static images from a perceptual point of view. He states that representations of motion – compared with reproductions of static scenes – have long been neglected. He therefore analyses techniques that have the ability to convey motion in static images. In order to judge the efficacy of representations of motion in the contexts of art, science, and popular culture he introduces the following criteria:

- ***Evocativeness.*** Indicates whether a motion's representation succeeds in convincing an observer of a sense of motion, that is, "is motion perceivable in the image at all?"
- ***Clarity of object.*** Indicates whether an observer can clearly identify either the object whose motion is represented or the object that designates one's own movement. Clarity of object is mostly indispensable in science whereas a suggestive image is often sufficient and even wished in art.
- ***Direction of motion.*** Indicates whether a motion's representation conveys its direction clearly. Here, the observer's previous experience can assist prediction, for instance, the image of a runner leaning forward probably indicates forward motion.
- ***Precision of motion.*** Indicates whether a motion's representation depicts the amount of motion properly, so that an observer is able to predict it. Precision of motion is crucial in the context of science, for instance, to communicate time-variant data.

Using these criteria CUTTING then analyses the following schemes for representing motion:

- ***Dynamic Balance or Broken Symmetry.*** Handcrafted paintings and sculptures of the natural poses of humans and figures in motion often show distortions in symmetry, such as hips and legs turning in a different direction than from shoulders and arms (see Figure 7-2a). These distortions let one realize postural motion.

- ***Multiple Stroboscopic Images.*** A single static image composed by a series of captured images, for instance, by multiple exposure photography, illustrates motion through several discreet instances in time (see Figure 7-2b).
- ***Affine Shear or Forward Lean.*** Shearing moving objects' shapes leaning them into the direction of motion as if "leaning them against a wind" illustrates the objects' locomotion in a static image (see Figure 7-2c).
- ***Photographic Blur.*** Photographs captured using a long-exposure time show blurred regions in those areas where parts of the targeted scene have moved during image acquisition.
- ***Image and Action Lines.*** Drawing multiple images of a moving object at a discreet point in time and additionally drawing multiple lines that align to the moving object's path or trajectory in the opposite direction to the direction of motion depict motion in static images (see Figure 7-1).

It turned out that *Image and Action Lines* fulfills all criteria well and, that it is well-suited for representing motion in static images.

Comic books are a visual medium that can communicate complex narratives without using words in a way that even children can understand. MCCLOUD [81] describe a wide variety of the symbolization and abstraction techniques used to generate sequential art in comic books. These include techniques to depict the motion of single objects and to illustrate noises and speeches bound to time. The principles of visual and sequential art include the following:

- ***Symbolization.*** Simplifies the perception of activities and events by an iconic language that abstracts from reality. The vocabulary of symbols includes arrows, strokes, bubbles, and signs.
- ***Action Lines.*** Indicate moving objects by well-placed strokes (see *Image and Action Lines*). In addition streaking the background indicates a moving camera.
- ***Ghost Images.*** Mark past, present, and future positions of objects by drawing multiple images of the original objects (see *Image and Action Lines*), e.g., the repeated contours of the baseball in Figure 7-3 illustrate its former positions.
- ***Visual Metaphors.*** Indicate non-visual phenomena like sound, speech, smell, tension, and feelings using symbols that are associated with a scene or story context.
- ***Panels.*** Frame single depictions to form the entities of a narration. Each panel can depict a single activity or event that contributes to the comprehension of a story.
- ***Closure.*** Represents the ability to reconstruct and conceive sequential processes and narrations based on depictions that omit transitional steps and show only discrete moments in different perspectives, e.g., arranged as a collection of key-frame panels.

Storyboard artists deploy similar techniques to visualize and illustrate the storyline of a movie as storyboards; KATZ [60] and BEGLEITER [8] present the underlying design principles. Storyboards are usually produced and reconsidered in collaboration with the director and the screenwriter of the motion-picture production; they provide a basis for discussions about the screenplay. Storyboards deliver a skeletal structure, which documents the set design and depicts the shooting directions for the story used when preparing the production. As effective diagrams for documenting, communicating, and discussing ideas, they let outside participants understand the layout of the story and set design. Common storyboard diagrams depict camera movements and frame part of the scene to communicate camera exposures efficiently. The following outlines some terms and principles frequently used for storyboarding shooting directions:

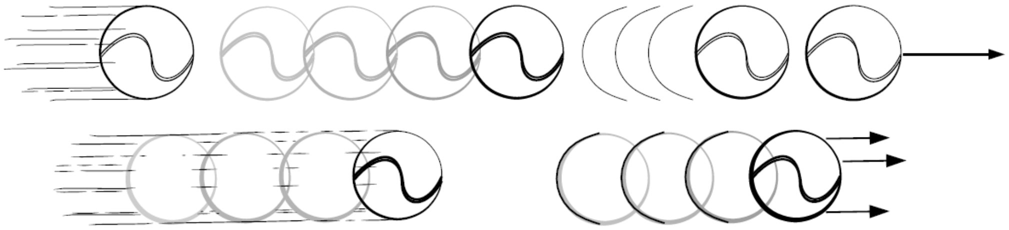


Figure 7-3: Speed Elements Depicting Motion. Speed elements, such as speed lines, repeated contours, and arrows, depict the baseball's motion. (Image by MASUCH ET AL. [80])

- ***Shot.*** Frames part of a staging; typically it indicates camera placement and narration instructions for the later production.
- ***Shot Flow.*** Represents the visual connection of a sequence of shots, whereby each shot can vary in size, aiming at a consistent spatial-temporal order. Shot sizes include medium and close-up shots.
- ***Medium Shot.*** Frames only half of a scene object, e.g., to capture an actor's gestures and body language.
- ***Close-Up.*** Frames a small part of a scene object in detail to position the viewer closer to it, e.g., to take a position for a dialog sequence.
- ***Crane Shot.*** Defines the uninterrupted movement of the camera in a predominantly vertical direction. At the beginning it establishes the environment towering above the scene (*establishing shot*) and then enters into details to direct attention from the general to the specific.
- ***Tracking Shot.*** Defines a flowing movement of the camera, tracking an object in a single shot or in a sequence of shots to visualize the varying composition of multiple story elements.

Traditional 2D hand-drawn animations use well-established techniques for conveying animations to make them entertaining [119][124]. THOMAS AND JOHNSTON identify and the term basic principles of traditional animations [119] and LASSETER postulates their importance in the field of 3D computer animations [71]. As a basic principle both describe the *squash-and-stretch* technique that squashes and stretches objects through motion to communicat their rigidity and mass. In a time-lapsed animation, the deformation of an object in each frame vividly depicts its dynamics: its velocity and acceleration. Thus, one can implement squash-and-stretch to depict an object's motion by generating multiple images of it showing forward lean. In a static image squash-and-stretch style depictions then maintain the clarity of moving objects and indicate their direction of motion [18]. To depict the ball's motion in Figure 7-4 squash and stretch has been implemented based on the technique described by CHENNEY ET AL. [14].

The visualization of motion in static images has rarely been discussed in the area of computer graphics. HSU AND LEE introduce *skeletal strokes* [54] for rendering speed lines [55]. *Speed lines* (a.k.a. action lines) streak away from the object in the opposite direction of moving. They thereby convey the objects' locomotion and its velocity in a way similar to motion blur but in a static image using expressive rendering. MASUCH ET AL. [79] present one of the first approaches in computer graphics that focuses exclusively on "presenting the motion of objects in computer generated still images" by way of *speed elements*. In addition to speed lines they repeatedly draw a moving object's contours depicting past movements, with arrows depicting future movements as well. Combining them in a single static image can depict past and future motion of objects (see Figure 7-3). Further stylization, such as applying line styles, achieves a hand-drawn impression.

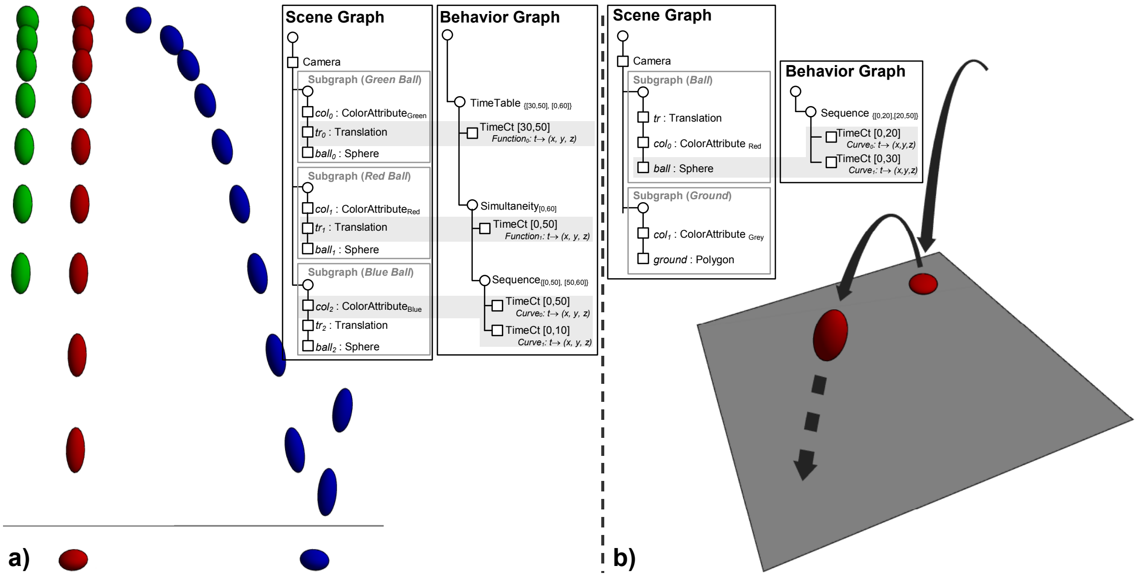


Figure 7-4: **Specifying Scenes and their Dynamics.** Squash and stretch applied as a traditional animation principle to bouncing balls. Scene and behavior graphs specify their animation and different time-layout strategies.

7.2 Dynamics Glyphs

NIENHAUS AND DÖLLNER introduce *dynamics glyphs* [94]. According to *abstract-graphical picture elements* [115] and speed elements [80], dynamics glyphs symbolize and depict dynamics, such as actions and time-related events, "in static images of 3D scenes". Dynamics glyphs include, for instance, action lines visualizing an object's locomotion, arrows indicating their future movements, ghost images depicting their past, present, or future positions, bubbles representing speech or thoughts, texts visualizing noises and frames capturing and portraying a dramatic moment as an instant in time to narrate a story.

Augmenting a static image with dynamics glyphs purposefully allows one to depict past, ongoing, and future processes of a certain situation or to convey non-geometric information such as tension, danger, and feelings. Furthermore, a sequence of augmented images even allows one to narrate a story line in a way similar to that of comic books and storyboards.

7.3 Specifying Scenes and Their Dynamics

First of all, it is shown how scenes and their dynamics are specified in the smart depiction system. These specifications represent the basis for all further functionality.

Specification Requirements

Hierarchical scene descriptions have a long tradition in computer graphics, and various scene graph libraries and scene description languages support them. In a typical scene specification 3D shapes, appearance attributes, geometric transformations, and environmental objects are arranged into a hierarchical structure. With these components, or scene nodes, developers can construct scenes composed of individual scene objects.

With respect to the specification of scenes, the following functional requirements are essential for the scene graph library:

- The scene specification must support a generic traversal operation, for example, to access each individual scene object, its components, and its scene graph context.
- The scene specification must allow one to assign (semantics-based) identifiers to scene objects and for defining complex scene objects.

With respect to the specification of dynamics, it is assumed that it must allow for the following requirements:

- Identifying which time-dependent changes are applied to a specific scene object.
- Determining the lifetime of each animation.
- Evaluating the scene specifications for any given point in time.

In the present approach, the scene graphs and behavior graphs as described by DÖLLNER AND HINRICHS [30] are applied. Both types of graphs are represented as directed acyclic graphs (DAG).

Figure 7-4b illustrates a scene graph that specifies a simple scene consisting of a sphere representing a ball and a polygon representing the ground. The graph branches into two subgraphs for specifying both the position (*tr*) and the color (col_0) of the ball and the color (col_1) of the ground.

Specifying Scenes using Scene Graphs

A *scene graph* specifies 3D scenes in a hierarchical way using scene nodes as building blocks [29]. In general, a scene graph library provides a collection of scene nodes, which model the structural and graphical aspects of 3D scenes. The following are descriptions of the most important categories of scene nodes:

- ***Groups.*** Build up the hierarchical structure and are used as the inner nodes of a scene graph. Examples are the *Branch Node*, which collects a number of subgraphs, and the *Switch Node*, which selects one out of the many subgraphs as an active child.
- ***Shapes.*** Specify geometric objects and are arranged typically as leaf nodes. Examples: *Box*, *Sphere*, *Cone*, and *PolygonMesh*.
- ***Transformations.*** Specify geometric transformations. The collection of transformation nodes along a path through the scene graph defines the transformation from the local to the world coordinate system. Examples: *Translation*, *Scaling*, *Rotation*, and *DirectionOfFlight*.
- ***Appearance Attributes.*** Specify properties and techniques that define the visual appearance of scenes and scene objects. Examples include *Color*, *Material*, and *Texture* as well as attributes used in the scope of non-photorealistic rendering such as *EdgeEnhancement*, *CartoonShading*, and *SketchyDrawing*.
- ***Environment Attributes.*** Specify properties of the scene's environment. Examples: *LightSource*, *PhongLightingModel*, *GoochLightingModel*, *ShadowCaster*, and *ShadowReceiver*.
- ***Non-Graphics Attributes.*** Provide the application-specific and semantics information within the hierarchical scene description. Examples: *Identifier* as a textual description of a subgraph, and *FilterTag*.

As a general mode of operation, only the nodes along the path from the root node to a specific node have an impact on that node and its components.

Scene Graph Rendering

For image synthesis the scene graph is traversed in pre-order. During the traversal, a graphics context manages hierarchically defined attributes. According to the rendering technique, scene graph rendering can imply multiple traversals of the scene graph. For example, non-photorealistic sketchy drawing produces several intermediate rendering results (see Chapter 6). Scene graph rendering is a critical real-time process; the scene graph library requires an efficient functionality to create and to modify scene graphs (e.g., adding new subgraphs) and scene nodes (time-dependent properties).

Scene Graph Inspection

The scene graph inspection represents a generic traversal function to report the structure and contents of a scene graph. The inspection allows one to retrieve the hierarchy layout (e.g., "Collect all nodes that represent a specific character having a specific Identifier"), browsing through the scene graph (reporting each node, its components, and types), and detecting dependencies between attributes and shapes etc. (e.g., "Which attributes apply to a given shape?", "Which shapes are affected by a given attribute?" etc.). Consequently, inspection is the best tool for interpreting scene graphs. The user can invoke the inspection function at any time, independent of the scene graph rendering.

Specifying Dynamics using Behavior Graphs

The *behavior graph* specifies the time-dependent and event-dependent aspects of scenes and scene objects. For a given scene graph, one or more behavior graphs may exist. Nodes of behavior graphs generally manipulate one or more nodes contained in the associated scene graph. The nodes used to construct behavior graphs are different from those for scene graphs.

The fundamental tasks of behavior graphs include the definition of the lifetimes of activities and the points in time of events. Activities and events specify time-dependent changes of in the properties of scene nodes. *Activities* take place during a defined, non-zero time interval, whereas *events* have no measurable duration because they take place instantaneously. Each node of the behavior graph provides its own time requirement, which represents the time needed to process the activity and event.

Layout of Time Flows

Time-group nodes, a major category of behavior graph nodes, hierarchically organize the time flow at a high level of abstraction similar to the specifications in storybooks. A time-group node calculates the lifetimes of its child nodes based on their time requirements and its own time-layout strategy. When a time-group node receives a time event, it checks which child nodes to activate or deactivate and then delegates the time event to its active child nodes. Specialized time-group nodes include the following:

- **Sequence.** Defines the total time requirement as the sum of the time requirements of its child nodes. It delegates the time flow to its child nodes in sequential order. Only one child node is alive at any given time during the sequence's lifetime.
- **Simultaneity.** Defines the total time requirement as the maximum of the time requirements of the child nodes. It delegates the time flow simultaneously to its child nodes. The simultaneity layout shrinks or stretches the time requirements of the child nodes or applies alignment strategies to the lifetime of the child nodes to fit the duration [30].
- **Time Table.** Defines for each child node an explicit time requirement. It manages activation and deactivation according to the child nodes' lifetime. For example, a time table can specify different starting times for individual objects in an animation.

Figure 7-4a illustrates a time-lapsed animation of three bouncing balls. Time layouts specify the lifetimes of each dynamic, for instance, a Time Table specifies different starting times for the green ball and the red and the blue balls.

Activities and Events

Having organized the overall time flow, *constraint nodes* let us specify activities and events. Essentially they associate a *time-to-value mapping* with the property of a scene node. For instance, constraint nodes can set up the position of an object by associating a time-to-vector mapping with the object's midpoint. Time-to-value mappings of the form:

$$f : Time \to Type$$

can implement a variety of mappings, such as mapping time to a constant value (*constant map*), to a value that results from linear interpolation of specified values (*linear map*), and to a value that results from calculating a point of a parameterized curve by interpreting time as a curve parameter (*curve map*).

A time-constraint defined as

$$tct : (f(Time), SceneNodes) \to SceneNodes$$

controls time-varying parameters of a scene node contained in the scene graph. Whenever a constraint node receives a time event during its lifetime, it calculates new parameter values, and assigns them to its constrained scene node. The generic class `TimeCt` takes care of most constraint variants.

In Figure 7-4a, time-constraint nodes (see the behavior graph) constrain translation nodes (see the scene graph) to specify the movement of the balls. A *function map*, which maps time to a value that results from a function call, controls the fall of the green and red balls taking gravity into account. In Figure 7-4b, two adjoining curves control the midpoint and, thus, the movement of the bouncing ball. They are processed in sequential order to form a single continuous trajectory.

Modifying Local Time Flows

Time-modifier nodes define *time-to-time mappings*, which can be used to alter the local time flow in behavior graphs. For instance, a *reverse modifier* inverts the direction of the time progress for its child nodes. Consider the bouncing ball in Figure 7-4b. Here, a reversal node could be used to invert the ball's direction of motion. Similar modifiers exist, such as repeating a time interval multiple times (*repeat modifier*) or defining a creeping time progress (*creep modifier*), that is, slowing down a progress in the beginning and speeding it up at the end of the time interval.

Behavior Graph Inspection

In an analogy to scene graphs, an inspection operation exists for behavior graphs, which allow one to examin the time flow and time mappings. For a given time interval we can reproduce activation and deactivation of behavior nodes, reproduce the results of a mapping, and identify the linkages of constraint nodes to scene nodes of the scene graph. Thus, the state of the 3D scene can be analyzed for a given point in time.

7.4 Assembling Dynamics Information

In the next step for generating smart depictions, we have to assemble dynamics information, that is, we must detect information about which nodes of the scene graph are affected by nodes in the behavior graph at any point in time.

First an inspection operation is applied to the scene graph to trace the path from its root node to a given scene node *node*. As result, we get the path set *P*(*node*) containing a sorted list of scene nodes with respect to their scene graph depth:

$$P(node) := \{ obj_k : obj_k \in path,\ k = depth \}$$

In particular, *P*(*node*) records all attributes and transformations that could potentially impact on the node *node*.

Then, an inspection of the behavior graph is invoked to analyze its time layouts. As a result, we determine the global lifetime of the time-constraint nodes. Let *tct* be a time-constraint node, then the analysis gives:

$$tct_{[t_0,t_1]}(f, obj) : active\ \forall t \in [t_0, t_1] \wedge inactive\ \forall t \notin [t_0, t_1]$$

Now, both results can be related to each other to determine the set of time-constraints that influence the properties of the scene nodes of *P*(*node*):

$$C(node) := \{ tct(f,obj) : obj \in P(node) \}$$

We can further derive a subset of *C*(*node*) containing time constraints that are active at a certain point in time t:

$$C_t(node) := \{ tct_{[t_0,t_1]}(f, obj) : t_0 \le t \le t_1, obj \in P(node) \}$$

Similarly, we can derive a subset of *C*(*node*) containing time constraints that are (anywhere) active in a given time interval [T_0, T_1]:

$$C_{[T_0,T_1]}(node) := \{ tct_{[t_0,t_1]}(f, obj) : obj \in P(node) \wedge [T_0, T_1] \cap [t_0, t_1] \ne \emptyset \}$$

Also taking into account the set *P*(*node*) of scene nodes, we can (1) evaluate the state or condition of a scene node at any point in time or time interval and (2) identify the types (e.g., translation) of scene nodes that contribute to a state change.

Because *P*(*node*) contains the transformation hierarchy, the trajectory of an object in 3D space can be easily determined by additionally sampling the set $C_{[T_0,T_1]}(node)$ at discreet points in time during its time interval. Position, velocity, and acceleration can then be used to depict an animation.

In conclusion, the tupel

$$D^{node}_{[T_0,T_1]} := (P(node), C_{[T_0,T_1]}(node))$$

represents the *assembled dynamics information* of a scene node during a time interval.

7.5 Characterizing Dynamics

Once the dynamics information have been collected for a scene object, one particular characteristic of the dynamics out of the many possible characteristics that we want to visualize in the final smart depiction has to be selected. The later symbolization process uses this information, which the user typically provides.

A *characteristic of dynamics* represents a token that classifies the kind of activity performed or event triggered by a scene object, the *depiction target*, in an informal way. In the case of a sphere as depiction target moving between two positions, we can declare a *path characteristic* to refer to the depiction target's trajectory.

Some basic characteristics of dynamics, which are elements of an extensible set of tokens, include the following:

- ***Path.*** Indicates a schematic description of the movement of a depiction target.

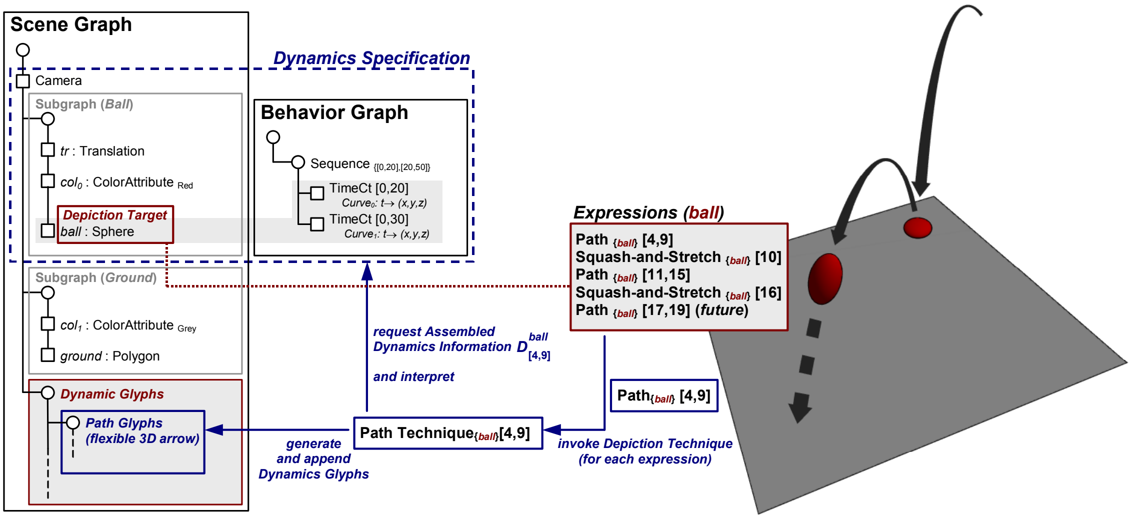


Figure 7-5: **System Overview.** The system overview (left) illustrates the general workflow for generating dynamics depictions. The scene and behavior graphs specify the ball's dynamics, which is the depiction target. Associating expressions to the ball allows one to model the representation of motion. A combination of squash and stretch and arrows as dynamics glyphs relate the bouncing ball's dynamics in the past, present, and future (right).

- ***Motion.*** Indicates a more natural and informal description of a movement in contrast to the path characteristic.
- ***Still.*** Indicates past, present, and future positions and orientations of a depiction target.
- ***Collision.*** Indicates a collision with other scene objects as an event related to a depiction target.

7.6 Symbolization and Depiction Techniques

Having assembled the dynamics information and chosen its dynamics, the system can now symbolize the dynamics. This process is encapsulated in *depiction techniques*, which implement specific characteristics of dynamics by mapping assembled dynamics information to dynamics glyphs. Technically, scene graphs specify dynamics glyphs, and these scene graphs link to the main scene graph as subgraphs for rendering.

For example, depiction techniques have been implemented for symbolizing path and motion characteristics. The *path technique* visualizes the trajectory of a movement; it constructs a flexible 3D arrow aligned to it and oriented towards the viewer (see Figure 7-5). For depicting motion, the *motion technique* generates motion lines, and includes additional strokes to provide a jittered appearance to make the motion easier to perceive (see Figure 7-6e).

A depiction technique requests assembled dynamics information for a given time interval and for a given depiction target as its main information source. Formally, we can define a depiction technique as a mapping of the depiction target's dynamics to a set of dynamics glyphs for the time interval $[T_0, T_1]$:

$$DepictionTechnique^{Characteristic} : D^{depiction\ target}_{[T_0,T_1]} \rightarrow DynamicsGlyphs$$

Figure 7-5 illustrates the path of a bouncing ball. The path technique generates arrows for the ball's trajectory. For this, it determines the depiction target's position at different points in time to reconstruct its path. The *still technique* constructs ghost images of the ball in a squash-and-

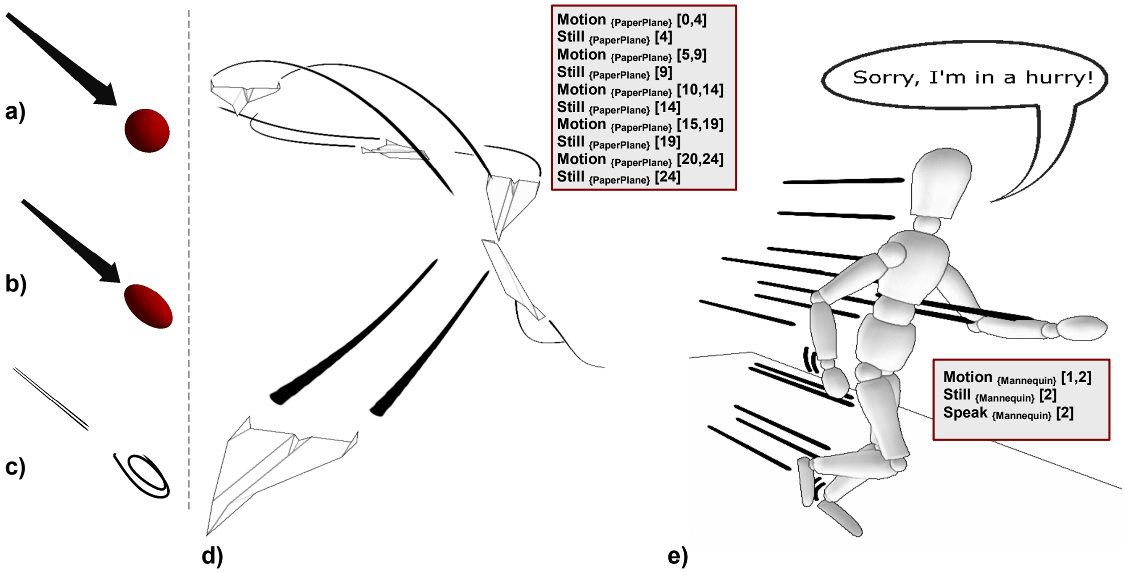


Figure 7-6: **Depictions of Dynamics.** (a-c) The system provides different kinds of symbols for a moving ball. (a) The path and the non-deformed ball visualize its motion in a motion-less way. The ball seems to rest at that point of travel. (b) The deformation of the ball using a squash-and-stretch technique depicts believable motion. (c) The sketchy depiction corresponds to an efficient drawing style and implies a fast moving ball. (d) Motion lines starting from the tips of the wings of the paper plane depict the paper plane's flight. (e) Expressions specify the dynamics depiction of the running, talking character. Motion lines are used for those parts of the character that move in the main direction of the motion and additional strokes for those that swing in opposite directions.

stretch style at discreet points in time. The depiction technique maps position, velocity, and acceleration to symbolize the ball in that traditional form.

Thus, with depiction techniques, the system maps triples consisting of the characteristics of dynamics, the time interval, and the assembled dynamics information to sets of dynamics glyphs.

7.7 Interactive Composition of Depictions

In practice, depicting dynamics represents a creative process and depends largely on the intentions and skills of the graphic designer. To give designers and artists as much control as possible, the system lets these depictions be interactively composed and customized. For this, an expression-like language is given, which allows users to invoke and set up depiction techniques.

Expressions let users create, store, apply, and configure dynamics depictions. In particular, users can directly set up parameters of depiction techniques to do the following:

- ***Control the visual appearance of dynamics glyphs.*** As an example, consider the still characteristic that indicates the positions and orientations of depiction targets as ghost images. The depiction technique can (1) simply render the depiction target (see Figure 7-6a), (2) render the depiction target in a squash-and-stretch style to visualize its velocity additionally by deformations (see Figure 7-6b), or (3) render a sketchy representation to mimic a hand-drawn illustration (see Figure 7-6 c).
- ***Control the composition of dynamics glyphs.*** For example, the collision technique symbolizes potential collisions between two given scene objects. The user can define the

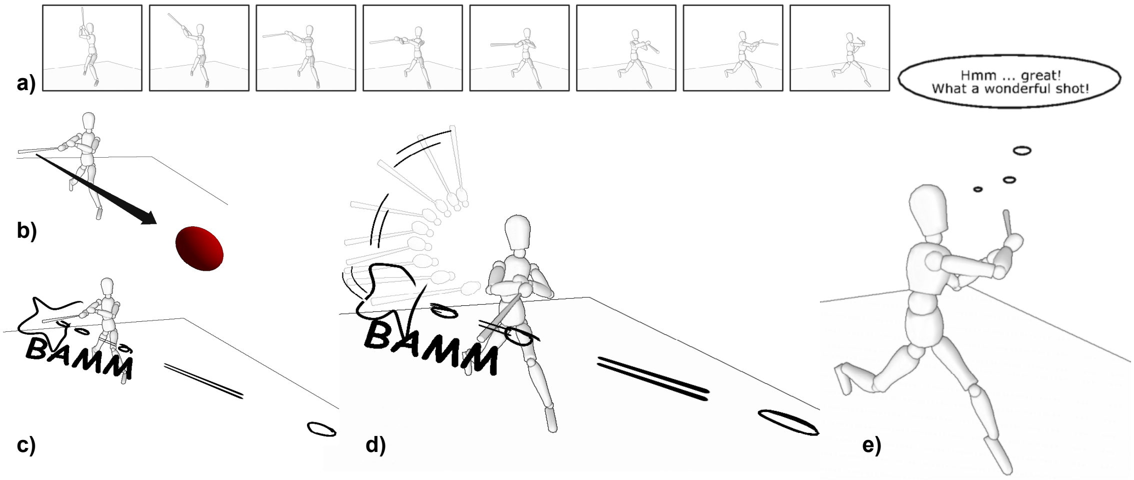


Figure 7-7: **Animation sequence showing a baseball batter hitting a ball.** (a) Single frames illustrate the original time-lapsed animation. (b) Depicting the batter when he hits the ball and the path of the ball after being hit. (c) Depicting the hit by symbolizing the collision and the noise that arises. The depiction shows the same action but in a sketchy style. (d) Narrating the batter's motion sequence, resulting in a more vibrant depiction. (e) Conveying ongoing motion by depicting the batter after realizing his excellent hit.

set of dynamics glyphs for visualizing collisions by rendering associated sounds as texts (see Figure 7-7).

- ***Control the composition of time.*** Defining what is past, present, and future is crucial for dynamics representations in images. For instance, the dashed arrow in Figure 7-5 effectively illustrates the path of the bouncing ball in the future. The user can provide temporal hints with expressions as an optional parameter.

In general, each expression requires the dynamics' characteristics that identify the depiction technique, the time interval to specify the period for depicting the dynamics, and optional parameters to configure the depiction technique.

7.8 System Overview and User Involvement

Figure 7-5 outlines the general workflow of the depiction system so far. To model the intended depiction of an objects' dynamics, the user selects a scene node in the scene graph as the depiction target and defines a set of expressions. Once associated with the depiction target, the system evaluates the set of expressions as follows:

1. For each expression, the system invokes the corresponding depiction technique, whereby technique choice is based on the characteristics of dynamics given by the expression.
2. The depiction technique requests the dynamics specification to retrieve the assembled dynamics information for the specified depiction target and time interval.
3. The depiction technique interprets the retrieved data and constructs dynamics glyphs.
4. The dynamics glyphs are linked to the main scene graph.

The pseudocode in Listing 7-1 illustrates the evaluation of a depiction target and its associated set of expressions. The system renders the 3D scene together with the dynamics glyphs. It doesn't render the depiction target itself because its picture is inessential and, more particularly, would interfere with the depictions of its dynamics.

In the present implementation, selecting a depiction target triggers the inspection of both scene graph and behavior graph. Depiction techniques can then be invoked and process in real-time, so that the user can interactively experiment with techniques using expressions and navigate the 3D scene. The sets of expressions (except for optional parameters) in the insets of Figure 7-6, Figure 7-5, Figure 7-8, and Figure 7-9 define the corresponding depictions.

Listing 7-1: **Mapping Dynamics to Dynamics Glyphs.** Expressions and depiction techniques allow one to map dynamics to dynamics glyphs.

```
procedure evaluate(SetOfExpression exprs, DepictionTarget obj) begin
  for all expr ∊ exprs begin
    /* Find depiction technique for requested characteristic */
    DepictionTechnique dt ← findTechnique(expr.Characteristic)

    /* Invokde depiction technique for generating dynamics glyphs
       based on assembled dynamics information and optional
       parameter */
    SceneNode dynGlyphs ← dt.depict($D^{depiction\ target}_{expr.TimeInterval}$, expr.Parameter)
    SceneGraph.append(dynGlyphs)
  end
end
```

7.9 Using Semantics for Depictions

Until now, we have merely considered scene nodes as depiction targets. However, a scene graph representation is sometimes not sufficient to define a depiction target unambiguously. That is, semantics information about scene objects must be available too. In particular, depictions of activities and events depend on semantics information because there are generally no obvious depiction techniques as in the case of depicting motion and path. For instance, we can’t generate and position meaningful bubbles symbolizing speeches and thoughts until the character and its head are explicitly defined (see Figure 7-6 and Figure 7-7).

Assigning Semantics to Scene Objects

Specifying scene objects with semantics information is subject to 3D scene modeling. Semantics information can be assigned to scene objects with a specialized attribute class, the *identifier*. Identifier attributes can form hierarchies to allow for hierarchical semantics descriptions for complex scene objects.

If a scene node contains an identifier, techniques looking for that kind of information will search in that node and its subgraphs; otherwise they will prune that node in the traversal. In this manner, the system can assemble a collection of scene nodes for one depiction target with specific semantics.

We define S as the set of scene nodes that contribute to a semantics description:

$$S := \{ node : node\ contribute\ s\ to\ semantics \}$$

The system assembles the dynamics information for each scene node. So,

$$D^{S}_{[T_0,T_1]} := \{ D^{node}_{[T_0,T_1]} : node \in S \}$$

provides all the assembled dynamics information that influence the depiction target with the specified semantics.

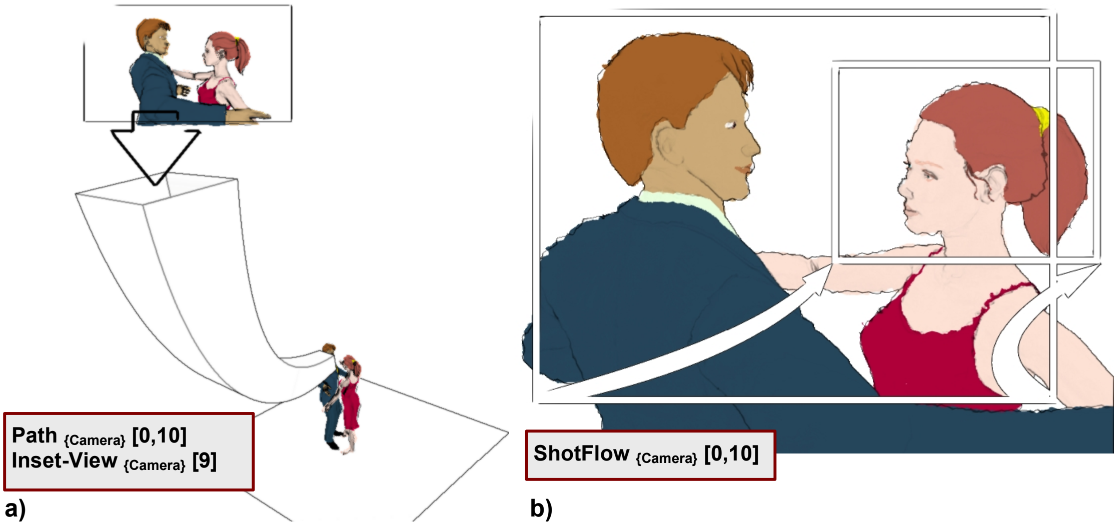


Figure 7-8: **Depictions based on Camera-Semantics.** Camera-related depiction techniques can visualize (a) crane shots or (b) close-ups in storyboard-like depictions. An additional inset view characteristic in (a) illustrates a medium shot of the scene; it was inspired by the long crane shot from Notorious [60]. Sketchy drawing has been used to generate the sketchy depictions (see Chapter 6).

An independent subgraph represents each of the characters in Figure 7-6 and Figure 7-7. The subgraph contains further subgraphs, each of which represents parts of the body such as the head, arm, or hand. In this case, S consists of those shapes that form the visible corpus of the character. Once animation data, such as motion capture data, has been assigned $D^{CHARACTER}_{[\tau_0,\tau_1]}$ provides all the dynamics information for the character. Consequently, depiction techniques can locate single parts of the body, identify their relationship to each other, or consider the character as the whole at any point in time to generate dynamics glyphs. So, by defining the character through a hierarchical composition of identifiers, we can narrate the batting sequence in Figure 7-7.

Semantics-Related Depictions

For scene graphs enhanced by semantics information we can refine depiction techniques and the characteristics of dynamics.

Semantics-Related Depiction Techniques

For a specific characteristic of dynamics depiction techniques can be implemented that convey that kind of dynamics more precisely for objects with specific semantics than a depiction technique implemented for a general scene object. For instance, a depiction technique that's specialized for camera semantics can symbolize the trajectory of a camera (path characteristic) as an extruded rectangular frame (see Figure 7-8a). Moviemakers often use this sort of depiction in storyboards to visualize a long crane shot [60]. In addition to the camera's position, an extruded frame encodes its viewing direction and alignment.

Another example is the depiction technique for the still characteristic of the paper plane semantics: it deforms the wings and endings of the paper plane under cross acceleration in a way similar to the deformation of a real paper plane. This can lead to a more dramatic appearance of the paper plane in a visual narration of its flight. The pseudocode in Listing 7-2 illustrates the modified selection procedure for semantics-related depiction techniques.

Listing 7-2: **Modified Mapping Procedure.** Invoking semantics-related depiction techniques.

```
procedure evaluate(SetOfExpression exprs, DepictionTarget obj) begin
  for all expr ∈ exprs begin
    /* Find semantics related depiction technique
       for requested characteristic */
    DepictionTechnique dt ← findTechnique(obj.Semantics,
                                          expr.Characteristic)
    SceneNode dynGlyphs ← dt.depict($D^{depiction\ target}_{expr.TimeInterval}$, expr.Parameter)
    SceneGraph.append(dynGlyphs)
  end
end
```

Semantics-Related Characteristics of Dynamics

Semantics information leads to a broader vocabulary for the characteristics of dynamics, that is, for a specific semantics new characteristics and the appropriate depiction techniques can be added.

For example, in cinematography a shot flow is clearly a characteristic of camera semantics. So, that characteristic can be added and its depiction technique using the design principles for cameras inspired by storyboard depictions can be implemented. Figure 7-8b illustrates a shooting direction from a medium shot to a close-up. Here, both frames indicate which part of the scene is visible when taking the shot at certain points in time. The arrows indicate the movement of the camera for taking the close-up. A lot of potential exists for exploring semantics-related depiction techniques to cope with the manifold ways of camera movements and illustrations of shooting directions deployed by storyboard artists.

Information Retrieval Functions

Depiction techniques request the function $D^{S}_{[T_0,T_1]}$ to retrieve encoded time-dependent data (see Figure 7-5), e.g., for the reconstruction of the transformation hierarchy. To facilitate the actual implementation of depiction techniques, *information retrieval functions* (such as the *hierarchy retrieval function*) are defined that search for and analyze the semantics-related data of a depiction target for a specific point in time. A *center retrieval function*, for instance, determines the center of a depiction target, which is needed to depict the object's trajectory. For a ball (or sphere) the center is likely to be the origin of its coordinate system in model space whereas a character's center isn't well defined. The center might be located in the character's geometrical bounding box. This is not appropriate because the box adjusts to its animated geometry. So, the hip as the character's center has been chosen. In conclusion, the path technique applies to both the ball and the character for constructing path glyphs. Thus, retrieval functions allow one to invoke depiction techniques for a broader set of semantics, respectively, depiction targets.

Core retrieval functions implement computational geometry algorithms. For instance, depiction techniques frequently require the extreme points of 3D geometries for constructing motion lines. An *extreme points retrieval function* can determine these points by evaluating the spatial arrangement of the geometries' vertices in strips, which are aligned to the object's moving direction [79][116]. In contrast, the extreme points retrieval function for the paper plane semantics provides wingtips that typically produce turbulence (depicted by motion lines in Figure 7-6d).

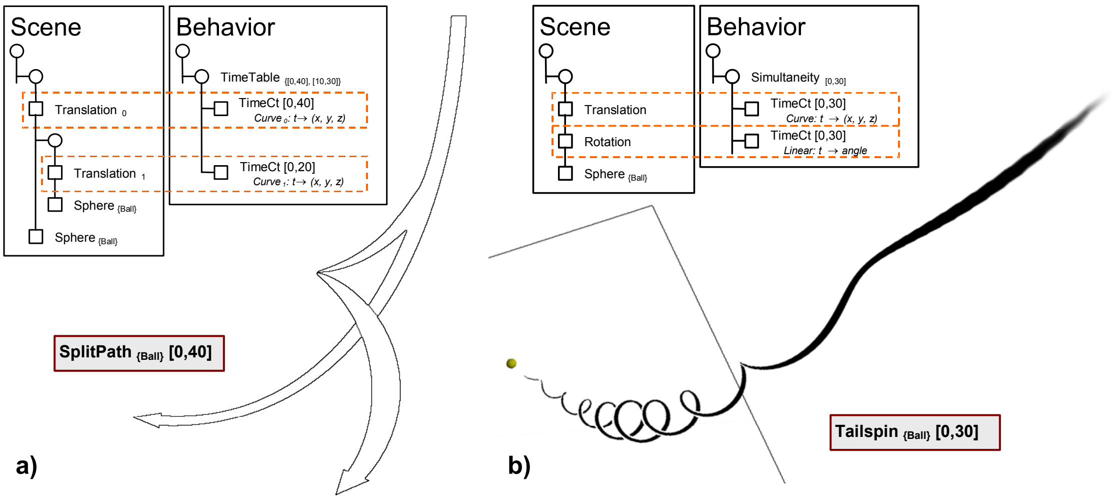


Figure 7-9: **Pattern-based Depictions.** Pattern-based techniques analyze scene and behavior graphs leading to advanced depiction techniques, such as (a) the split path or (b) the tailspin techniques.

For a character, retrieval functions considering human measurements [118] can determine information that isn't available at first glance. For instance, *facial measurement retrieval functions* can, based on the position and orientation of the head of a character, provide the position of the eyes, nose, and mouth even though they are not modeled explicitly. In this way, the *speaking technique* can align the cone of the bubble towards to the character's mouth in Figure 7-6e and Figure 7-10a. Thus, retrieval functions in combination with semantics information can provide beneficial information beyond their actual existence.

7.10 Pattern-Based Symbolization

Besides modeling depictions interactively using expressions, further analysis of dynamics by identifying patterns assists the process of symbolization. Pattern-based techniques allow one to determine relations

- in the composition of scene and behavior graphs,
- in the set of expressions, and
- among different dynamics that influence one another.

Thus, pattern-based techniques can give clues for producing depictions automatically and can enhance their comprehensibility.

Composition of Scene and Behavior Graphs

At a higher level of abstraction, the assembly of both scene and behavior graphs and the relationship between them can reveal patterns. In particular, animating those transformations that influence scene objects can lead to advanced characteristics of dynamics including the following:

- ***Tailspin.*** If a tailspin animates a scene, then a simultaneity group having two child nodes (one for constraining its position and one for constraining its rotation angle) encodes these dynamics in a behavior graph. The system then transfers a path characteristic to a *tailspin characteristic* to symbolize the dynamics with specific path glyphs (see Figure 7-9b).

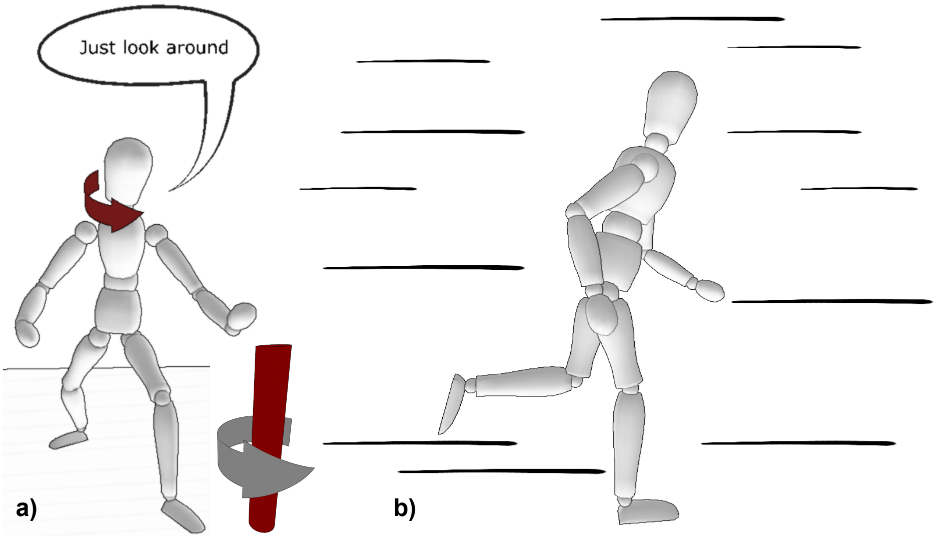


Figure 7-10: **Pattern-based Depictions.** (a) A bent arrow indicates a single rotation for illustrating a character turning his head to look around. (b) Motion lines streaking the background indicate the motion of both the camera and the character.

- ***Split path.*** If at least of the two scene objects that build up one depiction target follow the same trajectory in the beginning of an animation and then follow individual trajectories, their path splits smoothly into two. If we encounter a configuration of diverging paths in scene and behavior graphs, a *split path characteristic* can be applied (see Figure 7-9a).
- ***Turnaround.*** If a scene object rotates about an axis in model space, a constraint node in the behavior graph animates the rotation angle of a rotation transformation located directly in font of the object in its scene graph path (*P*(*node*)). Whenever we detect this composition, a *turnaround characteristic* for its motion can be created yielding a bent arrow aligned around the axis within a certain distance (see Figure 7-10a).
- ***Explode.*** The pattern indicating an *explosion characteristic* is similar to that of the split path but this time many scene objects of a single depiction target might abruptly change their direction of motion arbitrarily. Then, semantics-related explosion techniques either symbolize each launching part separately or produce a cloud of dust. They can also intensify the perception of the explosion by semantics-related sound using text.
- ***Expand/Collapse.*** An animated scale transformation that enlarges or scales down a single scene object can be interpreted by an *expansion/collapse characteristic*. If the scaling is located directly before the scene object, then the object pulses. Otherwise, if further transformations, e.g., translations, are located in-between, the object additionally moves in 3D space. The *expand/collapse technique* handles expansion and collapse differently. The technique symbolizes the expansion through multiple arrows starting at the scenes object's center and heading in different directions while increasing their width. They end at the estimated boundary of the enlarged object. In the collapse mode, the technique inverts the direction of the arrows, so that they point to the object's center. In the case of an assembly of diverging objects the explode characteristic can again be applied.

Set of Expressions

A set of expressions specified by the user can be subject to automatic enhancements. The system provides a join operation and a split operation to assist dynamics interpretation.

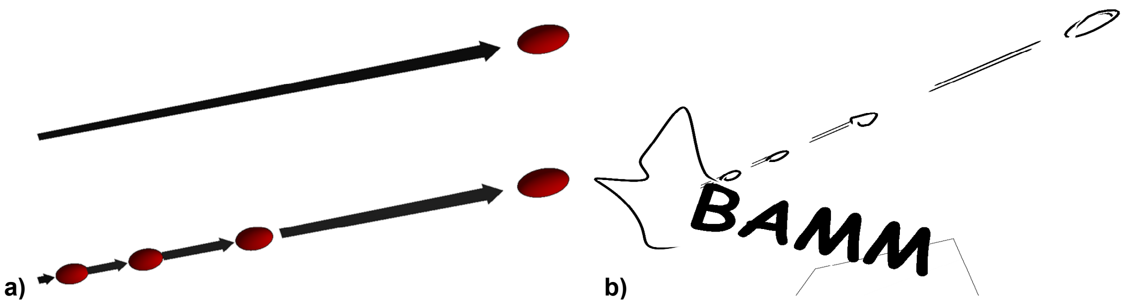


Figure 7-11: **Split-Operation.** (a) Splitting a path glyph depicts the ball's movement and its acceleration more expediently. (b) The sketchy depiction includes an additional causing event. The ball is catapulted and thus accelerated by an external force.

- ***Join operation.*** Sometimes multiple expressions that temporarily overlap can be depicted through specialized characteristics of the dynamics. The join operation scans the set of expressions and merges applicable expressions into single expressions to invoke advanced depiction techniques. For instance, a collision may occur during an object's motion. Visualizing both separately can produce dynamics glyphs that overlap in the depiction producing disturbing effects. Combining both enhances glyph constructions because a specialized depiction technique smoothly incorporates them for rendering.
- ***Split operation.*** The single expression of a long time interval can be split into several expressions because a more fine-grained schema might depict the dynamics more appropriately. To facilitate a split operation, the system queries the associated aspects of the dynamics in order to derive indicative information, such as velocity or acceleration. For instance, the motion characteristic of an accelerating object can be split into several expressions for motion to depict the dynamics in several time intervals and, thus, dramatize acceleration (see Figure 7-11).

Interacting Dynamics

The dynamics of depiction targets can influence the depictions of other objects' dynamics. For instance, well-placed motion lines allow one to distinguish fast from very fast movements. This is generally the case with a fixed camera. But if the camera is moving with the object, a *traveling shot characteristic* is more expedient. Here, the object remains focused while motion lines are then used for the background to depict the motion of both the camera and the object [81] (see Figure 7-10b). So, the relations of different dynamics influence depiction techniques in the whole. The pattern-based approach for symbolization helps resolve cases in which the dynamics influence one another.

7.11 Conclusions and Future Work

This chapter has presented a, automated depiction system for analyzing and symbolizing dynamics. Based on common scene and behavior specifications, the system produces smart depictions in a cost- and time-efficient way, and users can extend it with application-specific analysis and symbolization techniques. Here, non-photorealistic 3D rendering techniques achieves good results that come close to traditional and artistic works.

New designs of dynamics glyphs should be systematically implemented on top of the framework presented here. Future work might investigate which visual design of dynamics glyphs to use, for example, in the field of Virtual Reality and Augmented Reality applications. For the placement of dynamics glyphs, the layout, such as for frames and bubbles, can be further automated.

The semantics-based analysis and symbolization must be analyzed in more detail. In particular, all the camera-related depictions, which are relevant in the pre-production of a movie, need to be further optimized.

In addition, a pattern catalogue should be investigated. Although pattern-based techniques have been identified and the system can cope with the mapping of patterns to glyphs, pattern-based techniques, in particular interacting dynamics are a subject for future research. More techniques, patterns, and glyphs should be investigated for speech and sound as an interesting class of dynamics and an important category of multimedia contents.

The techniques for depicting dynamics presented here can enhance image quality even for standard interactive and animated computer graphics applications since they allow one outline certain activities, visually indicating events, or enhancing certain actors or objects. Depicting dynamics as a mostly automated process has great potential for rendering more than just 3D scenery into single images.

Chapter 8
Summary and Conclusions

The following statement has motivated the concepts in this thesis:

> *"Given a scene description, the number of rendering passes required by the rendering system – typically synthesizing and combining results of multiple rendering passes – will become less important in the long run but the essential questions will be whether the rendering system can produce visually compelling, comprehensible depictions."*
>
> *Jürgen Döllner*

Taking this as a starting point, the presented work has developed non-photorealistic rendering techniques and applications for generating depictions of 3D scenes and their dynamics.

The edge-enhancement algorithm transforms the concept of G-Buffers into the realm of real-time rendering. The edge map proves to be a flexible, generic, and collaborative tool for implementing a variety of edge-enhancement rendering techniques. Edge-enhancement allows applications and systems to display 3D scenes in an expressive and illustrative way and in real-time.

Blueprint rendering takes this approach a step further. It applies edge enhancement to several depth layers and overcomes a former limitation of image-space edge-based rendering, that is, it is capable of enhancing visible as well as occluded visually important edges. The vivid and expressive depictions composed by blueprint rendering enable viewers to understand the design and spatial assembly of complex aggregated objects as a whole.

Sketchy drawing uses the edge map as input for stylizing image-space edges. Sketchy and vague image representations of 3D models and 3D scenes let viewers understand a presentation as an early draft of a design and motivate them to participate in the ideas presented in it, to reconsider the design and to discuss improvements.

We found that the edge map approach applies well to image-based CSG rendering not only to illustrate a CSG model's shape using edge enhancement, but also to visualize its assembly, possibly in a sketchy manner.

Finally, the illustrative visualization of large-scale 3D city models demonstrates that the edge-enhancement algorithm can enhance scenes of high geometric complexity in real-time.

The second approach to comprehensible depictions presented in this thesis considers not only the geometric aspects of 3D scenes, but also their dynamics aspects. The smart depiction system developed for depicting dynamics analyzes scene and dynamics specifications. It maps the relevant dynamics to dynamics glyphs, and generates augmented images of 3D scenes using illustrative and expressive rendering. The resulting smart depictions can communicate the dynamics that are present in the 3D scene clearly, that is, the system enriches the information contents of the depictions of 3D scenery.

The work presented in this thesis shows that non-photorealistic rendering represents an enabling technology for conveying information comprehensibly using a human beings' innate ability to process visual information.

Illustrative and expressive visualizations of 3D city models emphasize a high perceptual and cognitive quality for effective communication. On the one hand, non-photorealistic rendering (i.e., edge enhancement and reduced color schemes) achieves visual clarity and conciseness in depictions of 3D city models. On the other hand, non-photorealistic rendering helps to communicate thematic information associated with 3D urban environments. Mapping the thematic information onto graphical attributes (e.g., line styles, procedural facade textures, and color tables) can convey this information in a subtle way.

Using the principles of visual art and visual narrations, the smart depiction system for depicting dynamics generates expressive visual contents that can visualize past, ongoing, and future activities and events taking place in and related to 3D scenes. The system relies on non-photorealistic rendering to generate smart depictions that come close to traditional drawings, such as storyboards. The storyboard-like depictions let non-experts, who are less comfortable with computer-graphics renderings, be involved in the design of sequential processes. Assisting the pre-visualization phase of a motion-picture production represents a significant application of the smart depiction system. For example, a series of depictions of dynamics derived from the given animations allow directors, set designers, and actors to actively participate, to discuss, to decide, and to reconsider sequential processes.

Depicting activities and events for visual communication raises a wealth of interesting issues for interdisciplinary work and need to be further explored.

The expression of a specific meaning in pictograms, signs, or decal information to advise and assist people represents a challenging task because it depends largely on the goal of the graphic designer. Future research could address the following:

- The possibility of generating meaningful pictograms, signs, and decal information automatically should be explored.
- What graphical representations of activities and events are most useful in a specific context should be explored.
- How much user involvement is essentially required to design the best composition of graphical representations should be explored.
- Suitable user-interfaces that allow graphic designers to customize pictograms, signs, and decal information in a cost and time efficient way could be developed.

Since the visualization of motion in static images has, on the whole, been neglected [18], depicting dynamics offers a large potential for future research in diverse contexts. For example, abstract image representations of all kinds of dynamics taking place in urban environments, such as traffic and the motion of crowds, should be explored. Illustrative 3D city models could assist in communicating dynamics in urban environments visually to finally synthesize depictions that convey more than just the projected 3D scenery.

BIBLIOGRAPHY

[1] Agrawala, M., and Durand, F., Guest Editors' Introduction: Smart Depiction for Visual Communication, *IEEE Computer Graphics & Applications: Smart Depiction Special Issue*, 25(3), pp.20-21, May/June 2005.

[2] Agrawala, M., Phan, D., Heiser, J., Haymaker, J., Klingner, J., Hanrahan, P., and Tversky, B.: Designing Effective Step-by-Step Assembly Instructions, *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH'03)*, pp.828-837, 2003.

[3] Agrawala, M., and Stolte, C.: Rendering Effective Route Maps: Improving Usability Through Generalization, *Proceedings of ACM SIGGRAPH'01*, pp.241-250, 2001.

[4] Agrawala, M., and Stolte, C.: A Design and Implementation for Effective Computer-Generated Route Maps, *AAAI Symposium on Smart Graphics 2000*, pp.61-65, 2000.

[5] Akenine-Möller, T., Haines, E.: *Real-Time Rendering (Second Edition)*, A K Peters, LTD, 2002.

[6] Apodaca, A. A., Gritz, L.: *Advanced RenderMan – Creating CGI for Motion Pictures*, Morgan Kaufmann Publishers, 2000.

[7] Appel, A.: The Notion of Quantitative Invisibility and the Machine rendering of Solids, *Proceedings of ACM National Conference*, pp.387-393, 1967.

[8] Begleiter, M.: *From Word to Image – Storyboarding and the Filmmaking Process*, Michael Wiese Productions, 2001.

[9] Benichou, F., and Elber, G.: Output Sensitive Extraction of Silhouettes from Polygonal Geometry, *Proceedings of 7th Pacific Graphics Conference*, pp.60-69, 1999.

[10] Blythe, D., Grantham, B., Kilgard, M. J., McReynolds, T., and Nelson, S. R.: Advanced Graphics Programming Techniques Using OpenGL, *ACM SIGGRAPH'99 Course Notes*, August 1999.

[11] Buchanan, J. W., and Sousa, M. C.: The Edge Buffer: A Data Structure for easy Silhouette Rendering, *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*, pp.39-42, 2000.

[12] Buchholz, H., Döllner, J., Nienhaus, M. and Kirsch, F.: Real-Time Non-Photorealistic Rendering of 3D City Models, *Proceedings of First International Workshop on Next Generation 3D City Models*, 2005.

[13] Cabarga, L.: *Dynamic Black & White Illustration – One Hundered Years of Line Art 1900-2000*, Art Direction Books, New York, 1995.

[14] Chenney, S., Pingel, M., Iverson, R., and Szymanski, M.: Simulating Cartoon Style Animation, *Proceedings of the 2nd International Symposium on Non-Photorealistic Animation and Rendering (NPAR) 2002*, pp.133-138, 2002.

[15] Claes, J., Di Fiore, F., Vansichem, G., and Van Reeth, F.: Fast 3D Cartoon Rendering with Improved Quality by Exploiting Graphics Hardware, *Proceedings of Image and Vision Computing New Zealand (IVCNZ)*, pp.13-18, 2001.

[16] Crow, F. C.: Shadow Algorithms for Computer Graphics, *Computer Graphics (ACM SIGGRAPH'77 Proceedings)*, 11(2), pp.242-248, 1977.

[17] Curtis, C.: Loose and Sketchy Animation. *ACM SIGGRAPH 1998 Conference Abstracts and Applications*, p. 317, 1998.

[18] Cutting, J. E.: Representing Motion in a Static Image: Constraints and Parallels in Art, Science, and Popular Culture, *Perception*, 31(10), pp.1165-1193, 2002.

[19] DeCarlo, D., Finkelstein, A., and Rusinkiewicz, S.: Interactive Rendering of Suggestive Contours with Temporal Coherence, *Proceedings of the 3nd International Symposium on Non-Photorealistic Animation and Rendering (NPAR) 2004*, pp.15-24, 2004.

[20] DeCarlo, D., Finkelstein, A., Rusinkiewicz, S., Santella, A.: Suggestive Contours for Conveying Shape, *ACM Transactions on Graphics (TOG), Proceedings ACM SIGGRAPH 2003*, 22(3), pp.848-855, 2003.

[21] Decaudin, P.: Rendu de scénes 3D imitant le style «dessin animé», *Rapport de Recherche 2919*. Université de Technologie de Compiègne, France, 1996.

[22] Decaudin, P.: *Modélisation par Fusion de Formes 3D pour la Synthèse d'Image – Rendu de Scènes 3D imitant le Style "Dessin Animé"*, Ph.D. Thesis, Université de Technologie de Compiègne, France, December 1996.

[23] Deering, M, Winner, S., Schediwy, B., Duffy, C. and Hunt, N.: The Triangle Processor and Normal Vector Shader: A VLSI system for High Performance Graphics, *Proceedings of ACM SIGGRAPH '88, Computer Graphics,* 22(4), pp.21-30, 1988.

[24] Deussen, O., Hiller, S., van Overveld, C.W.A.M., Strothotte, T.: Floating Points: A Method for Computing Stipple Drawings, *Computer Graphics Forum (Proceedings of Eurographics 2000)*, 19(3), pp.40-51, 2000.

[25] Deussen, O., Hamel, J., Raab, A., Schlechtweg, S., Strothotte, T.: An Illustration Technique using Hardware-based Intersections and Skeletons, *Proceedings of Graphics Interface 1999*, pp.175-182, 1999.

[26] Diefenbach, P. J.: *Pipeline Rendering: Interaction and Realism Through Hardware-based Multi-Pass Rendering*, Ph.D. Thesis, University of Pennsylvania, June 1996.

[27] Diepstraten, J., Weiskopf, D., and Ertl, T.: Transparency in Interactive Technical Illustrations, *Computer Graphics Forum (Proceedings of Eurographics'02)*, 21(2), C317-C325, 2002.

[28] Döllner, J., Buchholz, H., Nienhaus, M. and Kirsch, F.: Illustrative Visualization of 3D City Models, *Proceedings of SPIE – Visualization and Data Analysis 2005 (VDA 2005)*, 2005.

[29] Döllner, J., and Hinrichs, K.: A Generic Rendering System, *IEEE Transactions on Visualization and Computer Graphics*, 8(2), pp.99-118, 2002.

[30] Döllner, J., Hinrichs, K.: Object-oriented 3D Modeling, Animation and Interaction, *The Journal of Visualization and Computer Animation (JVCA)*, 8(1), pp.33-64, 1997.

[31] Döllner, J., Walther, M.: Real-Time Expressive Rendering of City Models, *Proceedings IEEE 2003 Information Visualization, Seventh International Conference on Information Visualization*, pp.245-250, 2003.

[32] Durand, F.: An Invitation to Discuss Computer Depiction, *Proceedings of Second International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000)*, pp.111-124, 2002.

[33] Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worely, S.: *Texturing & Modeling – A Procedural Approach (Third Edition)*, Morgan Kaufmann Publishers, San Francisco, 2003.

[34] Edwards, B.: *The New Drawing on the Right Side of the Brain*, Penguin Putnam, Inc., New York, 1999.

[35] Everitt, C., Kilgard, M. J.: Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering, *Online Published Paper*, NVIDIA Corporation, 2002.

[36] Everitt, C.: Interactive Order-Independent Transparency, *Online Published Paper*, NVIDIA Corporation, 2001.

[37] Fernando, R., and Kirgard, M. J.: *The Cg Tutorial – The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, 2003.

[38] Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F.: *Computer Graphics: Principles and Practice in C (2nd Edition)*, Addison-Wesley Professional, 2nd Edition, 1995.

[39] Freudenberg, B.: *Real-Time Stroke-Based Halftoning*, Ph.D. Thesis, University of Magdeburg, 2003.

[40] Freudenberg, B., Masuch, M., and Strothotte, T.: Real-Time Halftoning: A Primitive For Non-Photorealistic Shading, *Proceedings of 13th Eurographics Workshop on Rendering*, pp.227-232, 2002.

[41] Goldfeather, J., Molnar, S., Turk, G., and Fuchs, H.: Near Realtime CSG Rendering Using Tree Normalization and Geometric Pruning, *IEEE Computer Graphics and Applications*, 9(3), pp.20-28, May 1989.

[42] Gomes, J. and Velho, L.: *Image Processing for Computer Graphics*, Springer-Verlag, New-York, 1997.

[43] Gooch, A., Gooch, B., Shirly, P., and Cohen, E.: A Non-Photorealistic Lighting Model for Automatic Technical Illustration, *Computer Graphics (Proceedings of ACM SIGGRAPH'98)*, Orlando, FL, pp.447-452, July 1998.

[44] Gooch, B., Gooch, A.: *Non-Photorealistic Rendering*. A.K. Peters (2001)

[45] Gooch, B., Sloan, P. P J., Gooch, A., Shirley, P., and Riesenfeld, R.: Interactive Technical Illustration, *ACM Symposium on Interactive 3D Graphics 1999*, pp.31-38, 1999.

[46] *GPGPU Website*, www.gpgpu.org.

[47] Guha, S., Krishnan, S., Munagala, K., and Venkatasubramanian, S.: Application of the Two-Sided Depth Test to CSG Rendering, *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pp.177-180, 2003.

[48] Haeberli, P.: Paint By Numbers: Abstract Image Representations, *ACM SIGGRAPH Computer Graphics, Proceedings of ACM SIGGRAPH'90*, 24(3), pp.207-214, 1990.

[49] Hargreaves, S. and Harris, M. J.: Deferred Shading, *NVIDIA Online Published Presentation*, NVIDIA Corporation, 2004.

[50] Harris, M. J., Coombe, G., Scheuermann, T., and Lastra, A.: Physically-Based Visual Simulation on Graphics Hardware, *Proceedings of Graphics Hardware 2002*, pp.1-10, 2002.

[51] Harris, M. J. and Luebke, D. P.: GPGPU: General-Purpose Computing on Graphics Hardware, *ACM SIGGRAPH 2004 Course Notes*, 2004.

[52] Heidrich, W.: *High-Quality Shading and Lighting for Hardware-Accelerated Rendering*, Ph.D. Thesis, University of Erlangen, 1999.

[53] Hertzmann, A.: Introduction to 3D Non-Photorealistinc Rendering: Silhouettes and Outlines. *S. Green (Editor), Non-Photorealistic Rendering*, *ACM SIGGRAPH'99 Course Notes*, 1999.

[54] Hsu, S. C. and Lee, I. H. H., Wisemann, N. E.: Skeletal Strokes, *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology (UIST)*, pp. 197-206, 1993.

[55] Hsu, S. C. and Lee, I. H. H.: Drawing and Animation Using Skeletal Strokes, *Computer Graphics (Proceedings of ACM SIGGRAPH'94)*, pp. 109-118, 1994.

[56] Isenberg, T.: Capturing the Essence of Shape of Polygonal Meshes, PhD Thesis, University of Magdeburg, 2003.

[57] Isenberg, T., Freudenberg, B., Halper, N., Schlechtweg, S., and Strothotte, T.: A Developer's Guide to Silhouette Algorithms for Polygonal Models, *IEEE Computer Graphics and Applications*, 23(4), pp.28-37, July/August 2003.

[58] Isenberg, T., Halper, N., and Strothotte, T.: Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes, *Computer Graphics Forum (Proceedings of Eurographics)*, 21(3), pp.249-258, 2002.

[59] Kalnins, R. D., Davidson, P. L., Markosian, L., and Finkelstein, A.: Coherent Stylized Silhouettes, *Proceedings of ACM SIGGRAPH 2003*, pp.856-861, 2003.

[60] Katz, S. D.: *Film Directing Shot by Shot: Visualizing from Concept to Screen*, Michael Wiese Production, 1991.

[61] Kilgard, M. J. (Editor): *NVIDIA OpenGL Extension Specifications*, NVIDIA Corporation, Online Published, May 2004.

[62] Kilgard, M. J.: A Practical and Robust BumpMapping Technique for Today's GPUs, *Game Developers Conference*, 2000.

[63] Kirk, D.: *GPU Gems – Programming Techniques, Tips, and Tricks for Real-Time Graphic (Foreword)*, pp.xxvii-xxviii, Editor Randima Fernando, Addison-Wesley Professional, 2004.

[64] Kirsch, F.: *Entwurf und Implementierung eines computergraphischen Systems zur Integration komplexer, echtzeitfähiger 3D-Renderingverfahren*, Ph.D. Thesis, University of Potsdam, 2005.

[65] Kirsch, F., Nienhaus, M., Döllner, J.: Visualizing Design and Spatial Assembly of Interactive CSG, *Technischer Bericht, Hasso-Plattner-Institut an der Universität Potsdam,* Nr. 7/2005.

[66] Kirsch, F., and Döllner, J.: Rendering Techniques for Hardware-Accelerated Image-Based CSG, *Journal of WSCG'04*, 12(2), pp.221-228, 2004.

[67] Kirsch, F. and Döllner, J.: Real-time Soft Shadows using a Single Light Sample. *Journal of WSCG'03*, 11(2), pp.255-262, 2003.

[68] Krüger, J. and Westermann, R.: Linear Algebra Operators for GPU Implementation of Numerical Algorithms, *ACM Transactions on Graphics (TOG) (Proceedings of ACM SIGGRAPH 2003)*, 22(3), pp.908-916, 2003.

[69] Koenderink, J. J.: What does the Occluding Contour tell us about Solid Shape?, *Perception 13*, pp.321-330, 1984.

[70] Lansdown, J., and Schofield, S.: Expressive Rendering: A Review of Nonphotorealistic Techniques, *IEEE Computer Graphics and Applications*, 15(3), pp.29-37, May 1995.

[71] Lasseter, J.: Principles Of Traditional Animation Applied to 3D Computer Animation, *Computer Graphics (Proceedings of ACM SIGGRAPH'87)*, 21(4), pp. 35-43, July 1987.

[72] Lake, A., Marshall, C., Harris M., and Blackstein, M.: Stylized Rendering Techniques for Scalable Real-Time 3D Animation, *Proceedings of NPAR 2000: First International Symposium on Non-Photorealistic Animation and Rendering*, pp. 13-20, 2000.

[73] Li, W., Agrawala, M., and Salesin, D.: Interactive Image-Based Exploded View Diagrams, *Proceedings of Graphics Interface 2004*, pp.203-212, 2004.

[74] Lohan, F.: *Pen&Ink Techniques*, Contemporary Books, Inc., Chicago, 1978.

[75] Mammen, A.: Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique, *IEEE Computer Graphics and Applications*, 9(4), pp. 43-55, 1989.

[76] Mark, W. R., Glanville, R. S, Akeley, K., and Kilgard, M. J.: Cg: A System for Programming Graphics Hardware in a C-like Language, *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22(3), pp.896-907, 2003.

[77] Markosian, L., Kowalski, M. A., Trychin, S. J., Bourdev, L. D., Goldstein, D, and Hughes, J. F.: Real-Time Nonphotorealistic Rendering, *Computer Graphics (Proceedings of ACM SIGGRAPH '97)*, pp.415-420, August 1997.

[78] Markosian, L.: *Art-based Modeling and Rendering for Computer Graphics*, Ph.D. Thesis, Brown University, 2000.

[79] Masuch, M.: *Nicht-Photorealistische Visualisierungen: Von Bildern zu Animationen*. Ph.D. Thesis, University of Magdeburg, Shaker Verlag, 2001.

[80] Masuch, M., Schlechtweg, S., and Schulz, R.: Speedlines – Depicting Motion in Motionless Pictures, *ACM SIGGRAPH'99 Conference Abstracts and Applications, Computer Graphics Proceedings, Annual Conf. Series, (ACM SIGGRAPH'99)*, p.277, 1999.

[81] McCloud, S.: *Understanding Comics – The Invisible Art*, HarperPerennial, New York, 1994.

[82] Mitchell, L. J., Marwan, Y., Hart, E., and Hart, E.: Advanced Image Processing with DirectX® 9 Pixel Shaders, *ShaderX^2*, Wordware Publishing, 2003.

[83] Mitchell, J. L.: Real-Time 3D Scene Post-processing, *Game Developers Conference*, Online Published Paper, 2003.

[84] Mitchell, J. L., Brennan, C., and Card, D.: Real-Time Image-Space Outlining for Non-Photorealistic Rendering, *ACM SIGGRAPH 2002 Sketches and Applications*, 2002.

[85] Mitchell, J. L.: Image Processing with Direct3D Pixel Shaders, ShaderX: Vertex and Pixel Shaders Tips and Tricks, Wolfgang Engel (Editor), Wordware Publishing, 2002.

[86] Nadin, K., and Zakia, R. D.: *Creating Effective Advertising Using Semiotics*, The Consultant Press, Ltd., New York, 1994.

[87] Northrup, J. D. and Markosian, L.: Artistic Silhouettes: A Hybrid Approach, *Proceedings of NPAR 2000: First International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*, pp.31-37, 2000.

[88] Nienhaus, M. and Döllner, J.: Depicting Dynamics using Principles of Visual Art and Visual Narrations, *IEEE Computer Graphics & Applications, Smart Depiction Special Issue*, 25(3), pp.40-51, May/June 2005.

[89] Nienhaus, M. and Döllner, J.: Blueprint Rendering and "Sketchy Drawings", *GPU Gems II: Programming Techniques for High Performance Graphics and General-Purpose Computation*, pp.235-252, Editor Matt Pharr, Addison-Wesley Professional, 2005.

[90] Nienhaus, M. and Döllner, J.: Visualizing Design and Spatial Structure of Ancient Architecture using Blueprint Rendering, *Proceedings of VAST 2004 – The 5th International Symposium on Virtual Reality, Archaeology and Cultural Heritage*, pp.63-64, 2004.

[91] Nienhaus, M. and Döllner, J.: Sketchy Drawings, *Proceedings of the 3rd International Conference on Computer Graphics, Virtual Reality, Visualization and Interaction in Africa, ACM AfriGraph 2004*, 2004.

[92] Nienhaus, M. and Döllner, J.: Blueprints – Illustrating Architecture and Technical Parts using Hardware-Accelerated Non-Photorealistic Rendering, *Proceedings of Graphics Interface 2004*, pp.49-56, 2004.

[93] Nienhaus, M. and Döllner, J.: Sketchy Drawings – A Hardware-accelerated Approach for Real-time Non-Photorealistic Rendering, *Proceedings of the ACM SIGGRAPH 2003 Conference on Sketches and Applications*, 2003.

[94] Nienhaus, M. and Döllner, J.: Dynamic Glyphs – Depicting Dynamics in Images of 3D Scenes, *Proceedings of the Third International Symposium on Smart Graphics*, pp.102-111, 2003.

[95] Nienhaus, M. and Döllner, J.: Edge-Enhancement – An Algorithm for Real-Time Non-Photorealistic Rendering, *Journal of WSCG*, 11(2), pp. 346-353, 2003.

[96] Ostromoukhov, V.: Digital Facial Engraving, *Proceedings of SIGGRAPH'99, in Computer Graphics Proceedings, Annual Conference Series*, pp.417-424, 1999.

[97] Parker, J.R., *Algorithms for Image Processing and Computer Vision*, John Wiley & Sons, Inc., 1997.

[98] Peercy, M. S., Olano, M., Airey, J., and Ungar, P. J.: Interactive Multi-Pass Programmable Shading, *Proceedings of SIGGRAPH 2000, In Computer Graphics, Annual Conference Series*, pp.425-432, 2000.

[99] Perlin, K.: An Image Synthesizer, *Proceedings of ACM SIGGRAPH '85*, 19(3), pp.287-296, 1985.

[100] Praun, E., Hoppe, H., Webb, M., and Finkelstein, A.: Real-Time Hatching, *Computer Graphics (Proceedings of ACM SIGGRAPH'01)*, pp.579-584, 2001.

[101] Raskar, R. and Cohen, M.: Image Precision Silhouette Edges, *Proceedings of ACM Symposium on Interactive 3D Graphics 1999*, pp.135-140, 1999.

[102] Raskar, R.: Hardware Support for Non-photorealistic Rendering, *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp.41-46, 2001.

[103] Rossignac, J. R. and van Emmerik, M.: Hidden Contours on a Frame-Buffer, *Proceedings of the 7th Eurographics Workshop on Computer Graphics Hardware*, 1992.

[104] Rost R. J.: *OpenGL® Shading Language*. Addison-Wesley Professional, 2004.

[105] Sander, P. V., Gu, X., Gortler, S. J., Hoppe, H., and Snyder, J.: Silhouette Clipping, *Computer Graphics (Proceedings of ACM SIGGRAPH 2000)*, pp.327-334, 2000.

[106] Saito, T. and Takahashi, T.: Comprehensible Rendering of 3-D Shapes, *Computer Graphics (Proceedings of SIGGRAPH'90)*, 24(4), pp.197-206, August 1990.

[107] Schumann, J., Strothotte, T., Raab, A., and Laser, S.: Assessing the Effect of Non-photorealistic Rendered Images in CAD, *Proceedings of SIGCHI 1996 Conference on Human Factors in Computing Systems*, pp.35-41. 1996.

[108] Segal, M. and Akeley, K.: *The OpenGL® Graphics System: A Specification (Version 2.0 – October 22, 2004)*, Silicon Graphics Inc., 2004.

[109] Segal, M., Korobkin, C., can Widenfelt, R., Foran, J., and Haeberli, P. E.: Fast Shadows and Lighting Effects using Texture Mapping, *Computer Graphics (Proceedings of ACM SIGGRAPH'92)*, 26(2), pp.249-252, 1992.

[110] Shishkovtsov, O.: Deferred Shading in S.T.A.L.K.E.R., *GPU Gems II: Programming Techniques for High Performance Graphics and General-Purpose Computation*, pp.143-166, Editor Matt Pharr, Addison-Wesley Professional, 2005.

[111] Shreiner, D., Woo, M., and Neider, J.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4 (Fourth Edition)*, Addison-Wesley Professional, 2003.

[112] Sousa, M. C., and Prusinkiewicz, P.: A Few Good Lines: Suggestive drawing of 3D models, *Proceedings of Eurographics 2003: Computer Graphics Forum*, 22(3), pp.381-390, 2003.

[113] Strauss, P, S., Carley, R., An Object-Oriented 3D Graphics Toolkit, *Computer Graphics (SIGGRAPH `92 Proceedings)*, pp 341-349, 1992.

[114] Strothotte, T., Masuch, M., and Isenberg, T.: Visualizing Knowledge about Virtual Reconstructions of Ancient Architecture, *Proceedings of Computer Graphics International 1999*, pp.36-43, 1999.

[115] Strothotte, C. and Strothotte, T.: *Seeing Between the Pixels – Pictures in Interactive Systems*, Springer, 1997.

[116] Strothotte, T. and Schlechtweg, S.: *Non-Photorealistic Computer Graphics: Modeling, Rendering and Animation*, Morgan Kaufman, San Francisco, 2002.

[117] Teece, D.: Sable: A Painterly Renderer for Film Animation, *Proceedings of the ACM SIGGRAPH 2003 Conference on Sketches and Applications*, 2003

[118] Tilley A. R. and Henry Dreyfuss Associates: *The Measure of Man and Woman: Human Factors in Design*, John Wiley & Sons, 2001.

[119] Thomas F. and Johnston, O.: *The Illusion of Life: Disney Animation*, Disney Editions, 1995.

[120] Tufte, R.E.: *Visual Explanations*, Graphics Press, 1997.

[121] Upstill, S.: *The RenderMan Companion – A Programmer's Guide to Realistic Computer Graphics*, Addison-Wesley, 1992.

[122] Weiskopf, D. and Ertl, T.: Real-Time Depth-Cueing Beyond Fogging, *Journal of Graphics Tools*, 7(4), pp.83-90, 2003.

[123] Wiegand, T. F.: Interactive Rendering of CSG Models, *Computer Graphics Forum*, 15(4), pp.249-261, 1996.

[124] Williams R.: *The Animator's Survival Kit*, Faber & Faber, 2002.

[125] Winkenbach, G. and Salesin, D. H., Rendering Parametric Surfaces in Pen and Ink, *Computer Graphics (Proceedings of ACM SIGGRAPH 96)*, pp.469-476, 1996.

[126] Winkenbach, G. and Salesin, D. H.: Computer-Generated Pen-and-Ink Illustration, *Proceedings of ACM SIGGRAPH'94*, pp.91-100, 1994.

[127] Zander, J., Isenberg, T., Schlechtweg, S., and Strothotte, T.: High Quality Hatching, *Computer Graphics Forum (Proceedings of Eurographics 2004*), pp.421-430, 2004.

EHRENWÖRTLICHE ERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Dissertation ohne Hilfe Dritter und ohne Zuhilfenahme anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Potsdam, den 20. Juni 2005

Marc Nienhaus