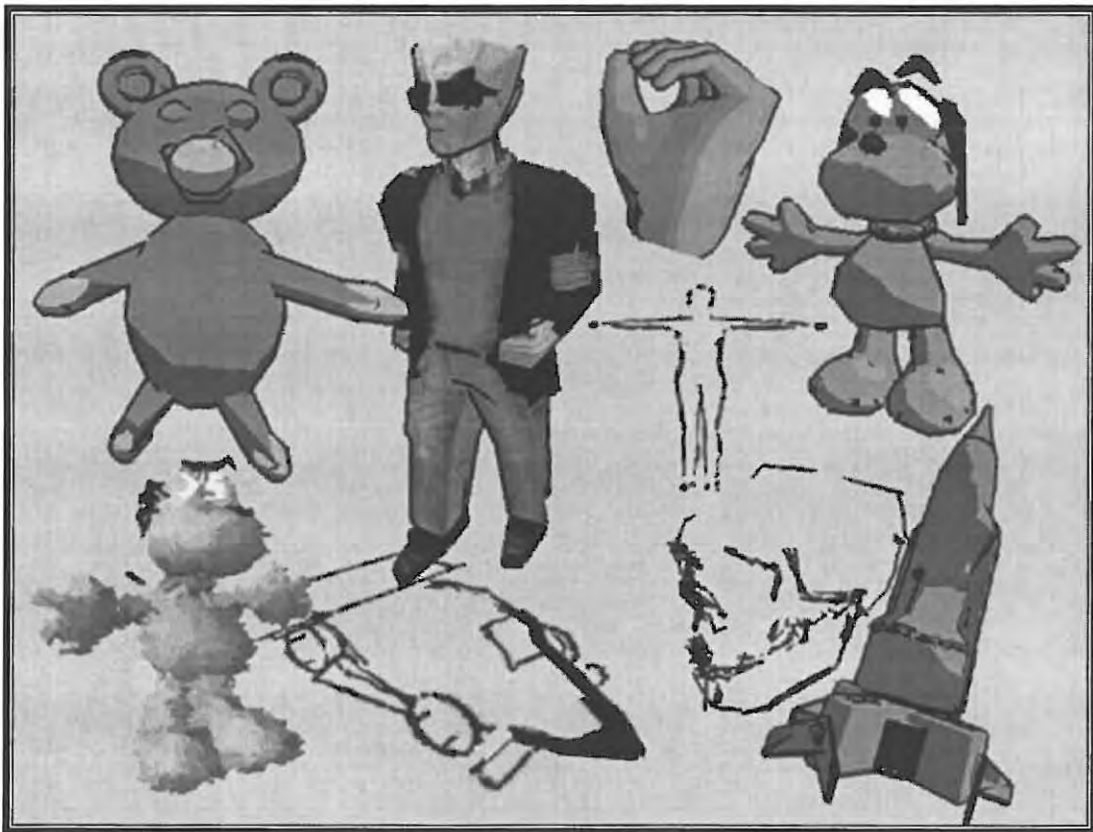


# **“Implementing Non-photorealistic Rendering Enhancements with Real-Time Performance”**

*by Holger Winnemöller  
in February 2002*



Submitted in fulfilment of the requirements for the degree of Master of Science  
of Rhodes University

## Abstract

We describe quality and performance enhancements, which work in real-time, to all well-known Non-photorealistic (NPR) rendering styles for use in an interactive context. These include *Comic* rendering, *Sketch* rendering, *Hatching* and *Painterly* rendering, but we also attempt and justify a widening of the established definition of what is considered NPR. In the individual Chapters, we identify typical stylistic elements of the different NPR styles. We list problems that need to be solved in order to implement the various renderers. Standard solutions available in the literature are introduced and in all cases extended and optimised. In particular, we extend the lighting model of the comic renderer to include a specular component and introduce multiple inter-related but independent geometric approximations which greatly improve rendering performance. We implement two completely different solutions to random perturbation sketching, solve temporal coherence issues for coal sketching and find an unexpected use for 3D textures to implement hatch-shading. Textured brushes of painterly rendering are extended by properties such as stroke-direction and texture, motion, paint capacity, opacity and emission, making them more flexible and versatile. Brushes are also provided with a minimal amount of intelligence, so that they can help in maximising screen coverage of brushes. We furthermore devise a completely new NPR style, which we call *super-realistic* and show how sample images can be tweened in real-time to produce an image-based six degree-of-freedom renderer performing at roughly 450 frames per second. Performance values for our other renderers all lie between 10 and over 400 frames per second on home-PC hardware, justifying our real-time claim. A large number of sample screen-shots, illustrations and animations demonstrate the visual fidelity of our rendered images. In essence, we successfully achieve our attempted goals of increasing the creative, expressive and communicative potential of individual NPR styles, increasing performance of most of them, adding original and interesting visual qualities, and exploring new techniques or existing ones in novel ways.

## CR Categories

- I.3.3 [Computer Graphics] Picture/Image Generation - Bitmap and framebuffer operations
- I.3.3 [Computer Graphics] Picture/Image Generation - Display algorithms
- I.3.4 [Computer Graphics] Graphics Utilities - Paint systems
- I.3.5 [Computer Graphics] Computational Geometry and Object Modeling - Boundary representations
- I.3.5 [Computer Graphics] Computational Geometry and Object Modeling - Curve, surface, solid, and object representations
- I.3.7 [Computer Graphics] Three-Dimensional Graphics and Realism - Animation
- I.3.7 [Computer Graphics] Three-Dimensional Graphics and Realism – Color, shading, shadowing, and texture
- I.3.7 [Computer Graphics] Three-Dimensional Graphics and Realism – Hidden line/surface removal
- I.3.7 [Computer Graphics] Three-Dimensional Graphics and Realism – Virtual Reality
- I.4.3 [Image Processing and Computer Vision] Enhancements – Geometric Correction

## **Additional Keywords**

3D Texturing; Brownian motion; Brush-stroke extensions; Comic, Cartoon, Coal, Pencil, Chalk, Hatching, Painterly rendering; Control-vertex acquisition & identification; Edge-fading; Importance-functions; Level-of-detail measure; Linear, bi-linear, tri-linear interpolation; Morphing; Natural media simulation; Non-photorealistic rendering; Perspective-correction; Phong lighting; Projective texturing; Realistic rendering; Real-time; Reference-image acquisition; Super-realistic rendering; Temporal coherence; Textured brushes; Tweening; Uncertainty-functions.

## **Acknowledgements and Legalities**

I would like to sincerely thank my supervisor Shaun Bangay for his loyal support, his cool vote of confidence and thought-provoking discussions; my girl-friend Danielle for her unconditional love and support; my parents for furnishing me with great values and always believing in me; Colin Dembovsky for years of excellent friendship; the Computer Science Department at Rhodes University for their friendliness and openness to new and exciting ideas; and Rhodes University for its vision, support, attitude and morale.

The copyright for this entire document and all of its content lies solely with Holger Winnemöller. Duplication of the whole document or parts thereof, by electronic, mechanical or other means is granted exclusively to Rhodes University and for educational purposes only, provided that no part of it is changed or omitted. Written permission of the author is required for all other purposes or by any other parties.

All models used in this project are publicly available (see Section 9.6 for source-references) in their original form and have subsequently been modified to suit our requirements.

All images and figures are created and copyright by Holger Winnemöller, except where explicitly stated otherwise.

# Table of Contents

<b>1</b>	<b>INTRODUCTION .....</b>	<b>1</b>
1.1	PROBLEM STATEMENT .....	1
1.2	A HISTORY OF GRAPHICS EVOLUTION ([103]).....	2
1.3	DEFINITIONS OF TERMS .....	2
1.3.1	<i>Photorealism vs. Non-photorealism</i> .....	3
1.3.2	<i>Real-time</i> .....	4
1.4	MOTIVATION FOR NON-PHOTOREALISTIC RENDERING .....	5
1.4.1	<i>Comic Style</i> .....	6
1.4.2	<i>Sketching</i> .....	7
1.4.3	<i>Painterly</i> .....	9
1.5	DOCUMENT STRUCTURE.....	9
1.6	SUMMARY .....	10
<b>2</b>	<b>RELATED WORK .....</b>	<b>11</b>
2.1	NON-PHOTOREALISTIC TECHNIQUES.....	11
2.1.1	<i>General</i> .....	11
2.1.2	<i>Comic Style</i> .....	12
2.1.3	<i>Sketching</i> .....	13
2.1.4	<i>Painting</i> .....	17
2.1.5	<i>Others</i> .....	19
2.2	TECHNICAL TERMS .....	20
2.2.1	<i>Object Description</i> .....	20
2.2.2	<i>Standard Vectors</i> .....	21
2.2.3	<i>Edges</i> .....	21
2.2.4	<i>Faces, Triangles and Polygons</i> .....	23
2.3	GENERAL TECHNIQUES .....	24
2.3.1	<i>Custom Clear Operation</i> .....	24
2.3.2	<i>Hidden Line Removal with Background preservation</i> .....	26
2.4	PERFORMANCE TESTING .....	27
2.4.1	<i>Set-up</i> .....	28
2.4.2	<i>Objects</i> .....	29
2.4.3	<i>Default Renderer</i> .....	31
2.5	SUMMARY .....	33
<b>3</b>	<b>COMIC STYLE .....</b>	<b>35</b>
3.1	INTRODUCTION .....	35
3.1.1	<i>Definition</i> .....	35
3.2	PROBLEMS.....	37
3.2.1	<i>Problem Statement</i> .....	37



3.2.2	<i>Implementation-specific Problems</i> .....	37
3.3	SOLUTION.....	38
3.4	STANDARD APPROACH.....	39
3.5	OPTIMISATIONS .....	40
3.5.1	<i>Geometric Redundancy</i> .....	40
3.5.2	<i>Face-Orientation Determination</i> .....	41
3.5.2.1	Perspective Correction .....	43
3.5.2.2	Qualitative Results for Face-Orientation Determination.....	47
3.5.2.3	Quantitative Results for Face-Orientation Determination .....	48
3.5.3	<i>Anti Aliasing</i> .....	50
3.5.4	<i>Relative Rotation</i> .....	50
3.6	EXTENSIONS.....	51
3.6.1	<i>Extending the Lighting Model</i> .....	51
3.6.1.1	Qualitative Results for Lighting model extension.....	53
3.6.2	<i>Using existing Colour Information</i> .....	56
3.6.3	<i>Silhouette Colour</i> .....	57
3.6.3.1	Quantitative Results for using Silhouette Colour.....	57
3.6.4	<i>Dealing with existing Textures (Multi-texturing)</i> .....	57
3.7	RESULTS .....	59
3.7.1	<i>Comparison of Approaches</i> .....	59
3.7.2	<i>Face-sorting vs. Displaylist in Extended Comic Renderer</i> .....	60
3.8	SUMMARY .....	61
4	<b>SKETCHING</b> .....	63
4.1	INTRODUCTION .....	63
4.1.1	<i>Definition</i> .....	63
4.2	PROBLEMS.....	65
4.2.1	<i>Problem Statement</i> .....	65
4.2.2	<i>Implementation-specific Problems</i> .....	65
4.3	SOLUTION.....	66
4.4	STANDARD APPROACH (RANDOM PERTURBATION SKETCHING).....	67
4.5	OPTIMISATIONS .....	68
4.5.1	<i>Level of Detail Optimisation</i> .....	68
4.5.1.1	Quantitative Results for Level of Detail Optimisations .....	68
4.5.1.2	Qualitative Results for Level of Detail Optimisations .....	69
4.5.2	<i>"Unconnected Triangles" Optimisation</i> .....	71
4.5.2.1	Quantitative Results for "Unconnected Triangles" Optimisation .....	71
4.5.3	<i>Recursive Algorithm Optimisations</i> .....	72
4.5.4	<i>Object-segmentation approach</i> .....	75
4.6	EXTENSIONS.....	78
4.6.1	<i>Pencil and Coal Sketching</i> .....	78

4.6.2	<i>Sketch shading</i> .....	82
4.6.2.1	Quantitative Results for Sketch Shading.....	88
4.7	RESULTS .....	89
4.7.1	<i>Comparison of Approaches</i> .....	89
4.8	SUMMARY .....	90
<b>5</b>	<b>PAINTING</b> .....	<b>93</b>
5.1	INTRODUCTION .....	93
5.1.1	<i>Definition</i> .....	93
5.2	PROBLEMS.....	94
5.2.1	<i>Problem Statement</i> .....	94
5.2.2	<i>Implementation-specific problems</i> .....	94
5.3	SOLUTION.....	95
5.4	STANDARD APPROACH (CONVOLUTION FILTERS).....	95
5.4.1	<i>Reference-image acquisition from 3D models</i> .....	97
5.5	OPTIMISATION (TEXTURED BRUSHES) .....	101
5.5.1	<i>The Brush</i> .....	101
5.5.1.1	Qualitative Results for Number of Brushes.....	105
5.5.1.2	Quantitative Results for Number of Brushes.....	106
5.5.1.3	Quantitative Results for Object Speed .....	107
5.5.1.4	Quantitative Results for Respawn Behaviour .....	110
5.5.2	<i>The Paint</i> .....	111
5.5.3	<i>The Technique</i> .....	111
5.6	EXTENSIONS.....	114
5.6.1	<i>Real-time Video Oil-painting</i> .....	114
5.7	RESULTS .....	116
5.7.1	<i>Qualitative Comparison of Approaches</i> .....	116
5.7.2	<i>Quantitative Comparison of Approaches</i> .....	117
5.8	SUMMARY .....	117
<b>6</b>	<b>SUPER-REALISTIC RENDERING</b> .....	<b>120</b>
6.1	INTRODUCTION .....	120
6.1.1	<i>Definition</i> .....	120
6.2	PROBLEMS.....	120
6.2.1	<i>Problem Statement</i> .....	120
6.3	SOLUTION.....	120
6.4	STANDARD APPROACH.....	121
6.5	OPTIMISATIONS .....	122
6.5.1	<i>Interpolation of Samples</i> .....	122
6.5.2	<i>Control Vertex Acquisition</i> .....	127
6.5.3	<i>Generalisation to 3D</i> .....	131

6.5.4	<i>Cubic Interpolation</i> .....	131
6.6	EXTENSIONS.....	132
6.6.1	<i>Motion Pictures and Special Effects (Morphing &amp; Tweening )</i> .....	132
6.6.2	<i>Web Advertisement (Tweening)</i> .....	132
6.6.3	<i>Extreme Low Bandwidth Video Conferencing (Tweening)</i> .....	132
6.6.4	<i>Facial Character Animation (Tweening)</i> .....	133
6.6.5	<i>Restoration of Video Material (Tweening)</i> .....	133
6.7	RESULTS .....	134
6.7.1	<i>Comparison of Approaches</i> .....	134
6.7.2	<i>General comments on Factors influencing Results</i> .....	134
6.7.2.1	Quality .....	134
6.7.2.2	Performance .....	135
6.8	SUMMARY .....	135
7	SYSTEM INTEGRATION.....	137
7.1	SOFTWARE CONSIDERATIONS.....	137
7.1.1	<i>Virtual Reality API</i> .....	137
7.1.2	<i>Graphics API</i> .....	139
7.1.3	<i>Operating Systems (OS)</i> .....	139
7.2	HARDWARE CONSIDERATIONS.....	143
7.3	AVAILABILITY ISSUES .....	144
7.4	SUMMARY .....	144
8	CONCLUSION.....	146
8.1	COMIC RENDERING .....	146
8.1.1	<i>General Problems and Solutions</i> .....	146
8.1.2	<i>Novel Concepts</i> .....	147
8.1.3	<i>Results</i> .....	148
8.2	SKETCH RENDERING .....	149
8.2.1	<i>General Problems and Solutions</i> .....	149
8.2.2	<i>Novel Concepts</i> .....	149
8.2.3	<i>Results</i> .....	151
8.3	PAINTERLY RENDERING .....	151
8.3.1	<i>General Problems and Solutions</i> .....	151
8.3.2	<i>Novel Concepts</i> .....	153
8.3.3	<i>Results</i> .....	153
8.4	SUPER-REALISTIC RENDERING.....	154
8.4.1	<i>General Problems and Solutions</i> .....	154
8.4.2	<i>Novel Concepts</i> .....	155
8.4.3	<i>Results</i> .....	156
8.5	EXTENSIONS AND FUTURE WORK .....	156

8.5.1	<i>Standard Improvements</i> .....	156
8.5.2	<i>Expansion of Definition</i> .....	156
8.6	CONTRIBUTION OF THESIS .....	157
<b>9</b>	<b>REFERENCES</b> .....	<b>159</b>
9.1	LITERARY REFERENCES .....	159
9.2	LIST OF ANIMATIONS .....	165
9.3	TABLE OF FIGURES .....	166
9.4	TABLE OF LISTINGS.....	169
9.5	LIST OF TABLES.....	170
9.6	SOURCES OF MODELS.....	171
<b>10</b>	<b>APPENDIX A - PHOTOREALISTIC TECHNIQUES</b> .....	<b>172</b>
10.1	PHOTOREALISTIC TECHNIQUES.....	172
10.1.1	<i>Ray-tracing ([101])</i> .....	172
10.1.2	<i>Radiosity ([101])</i> .....	174
10.1.3	<i>Hybrid (Ray-tracing and Radiosity [101])</i> .....	177
10.1.4	<i>Local Illumination Model</i> .....	179
10.2	REALISM ENHANCING TECHNIQUES.....	180
10.2.1	<i>Mapping Techniques</i> .....	180
10.2.1.1	Texture Mapping ([101]).....	181
10.2.1.2	Bump Mapping ([101]) .....	185
10.2.1.3	Environment Mapping ([101]).....	187

# 1 Introduction

We use this Chapter to give a crisp definition of the problems addressed by this thesis and the goals which we try to achieve. A short history of computer graphics helps establish a reference to the general graphics developments that have taken place up to this point. We define the key-words that form the foundation of this thesis and introduce the structure of this document.

## 1.1 Problem Statement

It is our goal to create or enhance Non-photorealistic (NPR) renderers that are able to perform in real-time for interactive purposes.

While the main impulse of computer graphics for the last few decades has been to create ever more realistic images in every shorter periods of time, a recent trend is moving in quite a different direction. NPR rendering is interested in creating images that look anything but photorealistic (hence the name) and relevant research results are published with increasing frequency. The key advantages of NPR rendering have been identified ([42], [47], [49], amongst others) as:

- Great creative and expressive potential
- Great communicative potential
- Possible render-speed improvements through abstraction of detail

We want to be able to harness this potential and use it where it can be most effective: in an interactive context. Only through interaction can the expressive and communicative possibilities of NPR rendering be tested to the fullest. To achieve smooth interaction, we have to supply a stream of images at interactive rates, i.e. in real-time (for a more detailed discussion of interactivity and real-time, see Section 1.3.2). With this in mind, we attempt to:

- Enhance the *performance* of existing NPR renderers
- Enhance the *visual quality* of existing NPR renderers, while remaining within real-time constraints
- Create new NPR renderers that perform in real-time

In order for our enhancements to be valuable and applicable, they have to achieve at least one of the following points with respect to the NPR style which they are meant to enhance.

- Increase its creative or expressive potential
- Increase its communicative potential
- Increase its rendering speed
- Expand its definition (i.e. add visual qualities which have not yet been considered)
- Employ new techniques or use existing techniques in new ways



It should be noted that while all of the above points mark desirable enhancements, some of these are easier to verify than others (i.e. an increase in rendering performance is more readily measured than an increase in communicative potential).

## 1.2 A History of graphics evolution ([103])

The development in computer graphics has in many ways paralleled if not induced similar developments in computer hardware. One of the earliest accounts of computer graphics was probably the *Whirlwind* at MIT in the 1960's, a set-up which used linear interpolation circuitry to produce very simple vector graphics on a conventional cathode ray tube oscilloscope. 3D to 2D transformations and clipping operations were so computationally expensive that the number of displayable objects and their complexity was extremely limited. In 1968 Evans and Sutherland responded by developing a vector pipeline and matrix multiplier to deal with these issues. A little later a cooperative effort between Sutherland and Sproull resulted in a clipping divider. Even with these advancements in place some serious problems remained: Storage of 3D objects was very expensive (as memory limitations were severe), specialised expensive hardware was necessary to update the display at a reasonable rate and only a few thousand vectors could be displayed before serious flickering indicated the limitations of the oscilloscope. In the late 1960's the so-called *direct view storage tube* was invented. Lines would be drawn and stored within this special type of CRT, alleviating the need for high-powered refresh facilities. Another advantage was that this new device could more easily be interfaced with mini-computers (as they were called at the time, despite their mostly enormous dimensions). The military, which up to that point had used physical scale models and armies of servos, became increasingly interested in using computer graphics for flight-simulation purposes. To become realistic enough for mission training though, the vector graphics of earlier days had to be replaced by solid-shaded geometry. As a result, the raster-scan device was invented. Since no screen buffer was available due to the prohibitive cost of memory, pixels had to be produced in scan-line order and also at a near constant rate even though realistically the processing load would vary greatly across each image produced. Even though complex and expensive, Watkins algorithm was used to deal with hidden surface removal. The slowly decreasing price of memory then allowed the implementation of a full-screen buffer (allowing the generation of pixels in arbitrary order) and the Painter's algorithm could be employed for hidden surface removal (although problems still prevail if objects are allowed to intersect). This problem was also overcome when further price-drops for memory allowed implementing a Z-Buffer approach. With the ever-ongoing advances in integrated circuit design several of the traditionally CPU intensive graphics operations like clipping, shading and now even transform and lighting operations have been offloaded from the CPU onto the graphics accelerator. Considering dozens of megabyte of RAM, dedicated processors capable of several hundred billion operations per second and sub-pixel accuracy in totally customisable lighting equations, the pioneers of computer graphics would probably stand in awe at the sight of today's possibilities.

## 1.3 Definitions of terms

This Section is dedicated to introduce and (where possible) firmly define key terms of *Non-photorealism* and *real-time*, which are used throughout this thesis.



### 1.3.1 Photorealism vs. Non-photorealism

The definition of an entire research subject as an exclusion of another is rather peculiar indeed and does injustice to the great potential of Non-photorealism. What it does do, is to show the stepchild-like role that Non-photorealism has played until very recently. In addition, the current definition is also flawed by its imprecise boundaries and we shall try to remedy this fact.



**Figure 1 – What you see is not what you get: "The Treachery of Images" (1929), René Magritte (1898-1967), Los Angeles County Museum of Art, Los Angeles, CA.**

We imagine a family party and two invited artists. One is a photographer and one is a scholar of Fine Arts with a drawing pad and a pencil. The photographer takes various photos of the family gathering, while the artist draws line-art drawings of the event. Most people would agree that the developed prints of the photographer are photorealistic (they are real in the sense that one can touch them or tear them up, but they are not reality in the sense that a photograph of a cake is not a cake. A very famous example of this notion was created by Magritte in Figure 1 – the caption reads "*This is not a pipe.*"). Similarly, many people would agree that the artist's drawing is Non-photorealistic, because through the use of deliberate pencil-strokes the artist has abstracted the visual reality to a degree that is not photorealistic anymore. The problem starts, when the photographer takes a photograph of the artist's drawing pad, or the artist through years of experience and perfected technique replicates a photograph in such perfection that it is indistinguishable from the actual photo. The first case seems easier at first: we are faced with a photorealistic reproduction of a non-photorealistic content. The second case seems unclear: the artist's drawing is so good that we cannot distinguish it from the actual photograph. But what if we didn't know about the photographer in the first case? Maybe the photo could be just a normal sketch on photographic paper. While it is not our intent to draw the reader into a philosophical discussion about reality, it should be clear by now that we need a more rigid definition for what we understand by photorealistic and non-photorealistic. Luebcke [46] has tried to find a unified approach to common NPR problems and states the following: "*Human Artwork typically consists of multiple separate strokes, ranging from dabs with a paintbrush to streaks of charcoal to lines with pen or pencil. Most NPR algorithms therefore cast rendering as a process of selecting and placing strokes. These strokes are usually placed and rendered with some randomness to imitate the unpredictability of the human artist...*". In our experience with the topic and in accordance with many researchers in the field, we therefore state that it is a matter of technique and abstraction more than anything else (these could be any of: visual outcome, content, style

and many others) that defines and distinguishes the current work on Non-photorealism. We would like to stress the word *current* in this context, because by the very nature of the original definition there could be any number of future rendering schemes that would force us to re-evaluate our current definition. In the light of this poor naming situation, some authors have already taken to referring to *artistic rendering* or *art-based rendering* instead of non-photorealistic rendering. We find that, although probably not ideal, the name *artistic rendering* reveals more about the involved process and allows for a certain level of qualitative measure (to the limited degree that one can measure artistic merit).

In conclusion, the following points are important in our understanding of NPR rendering:

- NPR artwork should exhibit an artistic quality. This is predominantly achieved through:
  - Abstraction (elimination of superfluous information)
  - An interpretation of reality through the eyes of the artist (or in our case the programmer, which is not necessarily mutually exclusive)
- NPR artwork should, in cases where it mimics human artwork, exhibit the same flaws and imperfections as human art.
- NPR should neither be defined solely as the opposite of other techniques, nor should it be limited to the few established NPR styles that are commonly investigated.

### 1.3.2 Real-time

The term *real-time* is a performance-measure and as it is used extensively throughout this thesis, we should express clearly what we mean by it. Firstly, when we speak about performance, we invariably mean the number of frames we can render in a given amount of time. Increasing performance means rendering more frames in less time. Some absolute definitions found during our research range from 70 frames per second to 4 seconds per frame. The problem with defining real-time in absolute terms is that it is actually application-dependent. Stewart [91] states that: “*Real time does not mean fast; it means that a system has timing constraints that must be met to avoid failure*”. In practice this means that the person operating the till in a super-market works with real-time constraints, even though he or she might only serve one customer every minute, but the super-market system would fail if he or she took an hour to serve a customer. Having gotten so far, we must identify what the time constraints in our system are. While it is certainly nice to produce pretty NPR images for people to look at, it is much more interesting to have them interact with these images, i.e. make the images react to a user’s input. We thus have to take a digression to investigate *Interactivity*.

For obvious reasons, interaction deals with the reaction to an action. Interactivity in turn deals with the timing issues involved in this process. We call the time between action and reaction the reaction-time or *latency* and make this a measure of interactivity. The latency of an interactive process can have a profound impact on a user. While some people might be slightly irritated by the long latency of their favourite word-processor, they can feel physically sick and experience vertigo, for even shorter latencies in an immersive environment. This is due to the fact that the effect of latency is strongly bound to the sense by which the reaction is perceived (e.g. even a small latency between visual stimulation and

corresponding stimulation of the sense of balance can produce almost instant motion-sickness). Our NPR scenes are to be perceived by the visual system of the human brain, so we will have a very brief look at relevant issues.

There are two main factors that are important for humans to perceive motion from animation (which is necessary to convert still images into interactive scenes). One is called *persistence of vision* and is largely physio-optical and the other one is called the *phi phenomenon* (attributed to Max Wertheimer, 1880-1943) and is psychological ([38]). The former describes the ability of the retina to retain optical information for a short period of time even after the stimulus is removed (in the order of hundreds of milliseconds). This means that we can see the a complete image on a television set instead of a very quickly moving electronic beam. While this is important to eliminate flickering it bears no relevance with respect to perception of motion (in the sense that the update of the screen-content and the update of the display-device can generally be decoupled). The latter phenomenon allows our brain to fuse discrete still images into fluid motion and is of greater relevance. As we produce discrete images of our NPR scenes as part of a temporal sequence, we want the observer to fuse these images into a continuous flow of motion. Depending on the environment, the scene under consideration and the distance between related objects in consecutive frames this fusion occurs at between 4 and 10 frames per second (i.e. a latency between 100-250 milliseconds. See [99] and [66]). This is not to say that motion cannot be deduced from experience or expectation at lower frame-rates, but this would have to happen consciously and with effort.

We now have narrowed our definition to the range in which motion from discrete images can be perceived naturally, which is necessary in an interactive animation context. Any further restriction will depend on the application context, as discussed above, i.e. what the user will do with the perceived motion (e.g. just observe, or use the perceived information in decision-making processes, which in turn might be more or less constrained by real-time considerations).

In compliance with various other authors, we fix our definition of real-time at the upper limit of the above range, at 10 frames per second, and define this rate the minimum acceptable performance for any of our renderers.

## 1.4 Motivation for Non-photorealistic rendering

Even though we deal in this thesis almost exclusively with technical issues of NPR, there exist various non-technical NPR-related issues that have been studied. For example the question, why do we need NPR? As the answer to this question might not be immediately obvious, but bears heavily on the relevance of this thesis, we will use this Section to motivate the most common NPR styles, which are:

- Comic or Cartoon
- Sketching (including Hatching)
- Painting

### 1.4.1 Comic Style

Many authors ([75], [71], [72] & [49] to name but a few) agree that the history of comics is as old as human history itself. They argue that cave drawings of approx 3000 B.C. (Figure 2a) as well as the sequential art (as comics are sometimes referred to) of historical pieces like the Bayeux Tapestry (Figure 2b), dating hundreds of years back, are some of the prominent examples of early comics in the course of history.



Figure 2 - Historical Comics (taken from [72]): a) Palaeolithic painting in the Altamira Cave, Spain; b) A Section of the Bayeux Tapestry (William's Army attacks the castle of Divan)

The comic as we know it today has mainly taken shape in the last century and its development was largely influenced by its perceived role in society, its commercial viability and technological limitations. Illustrated books like the popular “*Max and Moritz*” of Wilhelm Busch were targeting children as their main audience and for many years to come comics were frowned upon as *kid's stuff*, standing in the no-man's land between distinguished literature and acquired art. Only after newspaper sales increased due to regularly featured comics and research showed that comics and cartoons are able to convey educational and instructional messages much more clearly than words or photographs, did critics take serious notice of the new medium. One such research was undertaken by the US Defence Department and concludes that of a variety of possible military training manuals (plain text, illustrated text, text with photos and comic strips), the comic strip “*proved most effective in getting the requisite information across. The research showed that it is possible in comics to convey enormous amounts of information in a very limited space.*” [76].

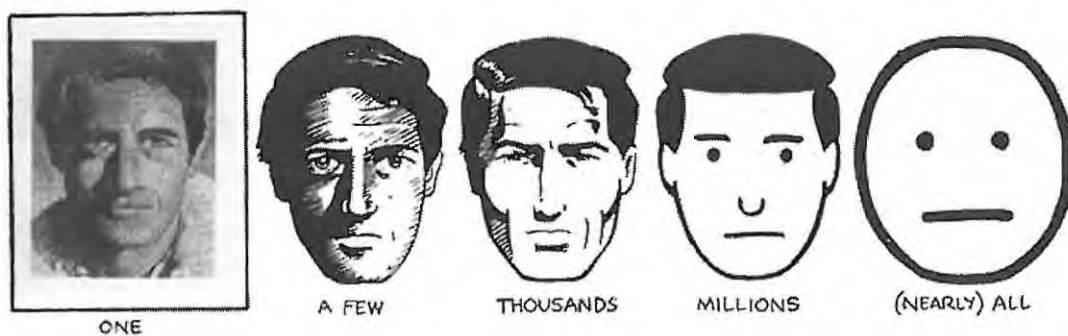


Figure 3 - Abstraction as a tool for self-identification [49]



The images in Figure 3 show increasing levels of abstraction from almost photographic to iconic. Below each picture is shown how many people could be said to resemble the image. This means that as the images become more abstract, more and more people are able to identify themselves with the image (or the portrayed character – McCloud [49] argues that this character-identification is even more pronounced in children, which he concludes, is the reason why comics are so popular with them). In the same way the level of abstraction greatly influences what information and detail the viewer observes. On the left-most picture one might wonder for example if the picture taken was for a passport and if the character is about to embark on an important journey and what the journey's purpose might be. As the level of abstraction is increased, the level of detail is decreased. The last image merely shows what most people will perceive as a face: No Age, no Gender, no Race, no possible resemblance to a real person – which is why it can be used to represent a face-bearer of any Age, Gender or Race. The absence of concrete information in this case does not deny its existence. A very commonly found application of this are the male/female figures on bathroom doors. They are extremely abstracted. Were they to be photographs of actual people one would most likely have thoughts like: “*Is this a missing person?*”, “*Does this Person live here?*” or similar. So if we want to convey the idea of a *man* (as opposed to a college student having a passport picture taken) we fare better to use some kind of comic-style abstraction. In addition to this, work by Hoffman [35] suggests that the process of seeing is an extremely involved one, which works in numerous stages (edge-detection, shape-recognition, object-recognition to summarise just the major steps). The inherent abstraction that comic-style imagery provides thus naturally helps in the recognition process by eliminating superfluous information and thus pre-processing information for the viewer.

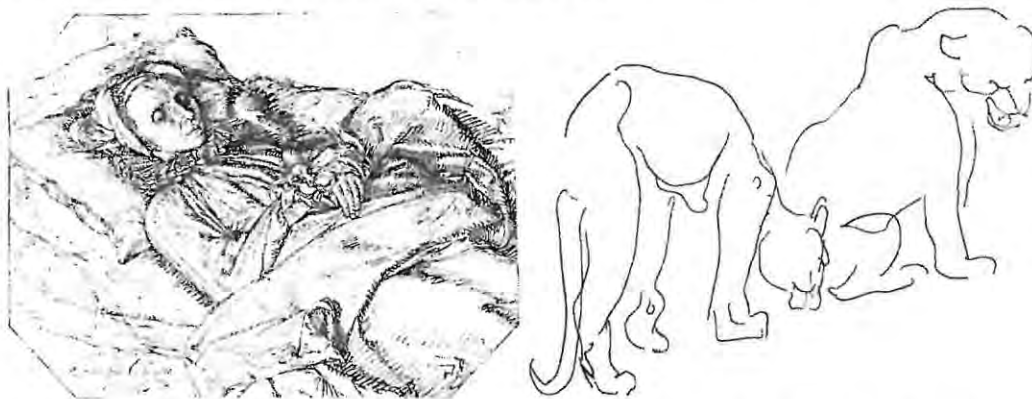
Comics derive their attractiveness largely from two factors: they are simple (or at least simplified versions of reality – even if they have a very complicated plot) and they are universal (through the use of simplistic depiction, language barriers as well as age barriers can easily be broken). They have been publicised in various print-media and connected to the motion picture industry from its very beginnings. It seems only a natural progression to merge the creative potential of computer graphics with the expressive potential of comic rendering. This development has in fact already started and by now computers play an important role in classic cartoon animation production. From creating backdrops for movies such as *Futurama* [19], *Dilbert* [1] or *Rugrats - the Movie* [100] to full-length features like *Final Fantasy* [13] or *Titan A.E.* [98], computers are an essential tool in increasing performance and productivity in today's cartoon industry. While this is true, the application areas of computers so far mainly cover the design- and off-line rendering stages.

Considering the huge expressive potential of comics, we are very interested in its use in Virtual Environments and therefore need to uncover ways of producing images rendered in a recognisable comic-style in real-time.

#### 1.4.2 Sketching

While the creative process of producing a sketch may seem hasty and mostly effortless, there are indications that sketching communicates most intimately with the most basic form of visual processing in humans. Hoffman [35] describes in his book (one of the Chapters of which is appropriately called

*“Inflating an Artist’s Sketch”*) the very involved processes that allow humans to see (i.e. to detect shape and motion from images of various compositions and situations). He devises basic rules which are applied to images by the brain and according to which object shapes and boundaries are detected. Many of these rules rely on edge-detection processes in the eye and help identify the most basic geometry underlying a given object. Attributes like shading, texture etc. are secondary clues in object identification (i.e. a cow with a woollen sweater and leather pants would still be a cow; even though a rather peculiar one). One might conclude from this that an inherent knowledge of these rules (if not in explicit form then at least in the creative process: we draw lines until we ourselves recognise the object; i.e. until the most basic recognition criteria are met) is applied when generating sketches and that this is the reason why they are so helpful in conveying a definite (and purposefully limited) set of information. Figure 4 demonstrates the enormous expressive potential of sketching. Figure 4a) is delicate, detailed and deliberate with every single stroke, capturing the intensity and finality of the moment. Figure 4b) on the other hand is minimalistic to the extreme, no superfluous lines clutter the image, yet the grace and poise of the animals is depicted with life-like precision. No physically correct light interaction has to be modelled, nor millions of fur-particles rendered in order to convey the intricate relationship between these creatures. Abstraction proves to be a powerful tool in extracting relevant information.



**Figure 4 - Two extreme Examples of Sketching: a) “Woman on Her Death Bed” ([24]);  
b) “Two Lionesses” ([5])**

As the visual form of sketches are much less rigorously defined in comparison to, for example, cartoons they are available as an expressive medium to a much larger audience. Almost all people will in some time in their life produce some form of sketch, be this to pass time on the telephone, to give directions to a friend or illustrate complex relationships to colleagues. More advanced sketches are used throughout professional and artistic communities to produce sketches for fashion-design, paintings, sculptures or as an end in itself. Yet more sophisticated and extremely formalised sketches can commonly be found as technical illustrations in manuals, medical books and architectural design. If we care to look, we can find sketches in many aspects of our daily lives. This familiarity combined with the abstractive and creative potential makes sketch rendering a prime candidate for many educational or design applications. Automating the sketching process with the help of computers is therefore a logical and necessary step.



### 1.4.3 Painterly

The motivation of painting from a psychological or sociological point of view is much more difficult than for cartoon or sketching, because only very few people interact with paintings on a day-to-day basis and they are not very useful to most people as they are usually expensive and take a long time to produce. As paintings have been used since the beginning of man's creative awareness to preserve for generations to come the moods and daily lives of man's existence it would be unthinkable to exclude them from any complete discussion on NPR techniques.

What painting so far cannot do is change. They are cast in paint and imprisoned on canvas. If people can be so fascinated by still paintings, even though they are not *useful* in many practical terms, we are interested to see what an effect animated paintings can have on the observer. If paintings can be produced in real-time and at sufficient quality, people can not only look at paintings, but even interact with them or watch them tell stories. These incentives alone should warrant an investigation into implementing a real-time painterly renderer.

## 1.5 Document Structure

This first Chapter helps us set the scene for our thesis. A definitive problem statement is expressed, clearly stating the fundamental problems that are addressed in this work. A short history of graphics evolution explains the origins of computer graphics so that the current state of art, discussed later, is placed into perspective. We follow this with motivations for each of our renderers and define the terms forming the key-words of the title of this thesis.

Chapter 2 studies the current state of the art in NPR research and introduces general terms and techniques used throughout this thesis. Furthermore, the testing procedure for all our performance tests is explained, so that it can be used as a reference for our individual NPR Chapters.

The following four Chapters introduce in turn each of our NPR renderers. We always start with a definition of style or technique and from this form a general problem statement, which is then expanded to include implementation-specific problems. A commonly accepted solution in the form of a standard approach is then presented and based on this enhancements are made which are ensured to work in real-time. With each of the modifications we make to the standard solution, we immediately provide qualitative and/or quantitative results that show the effects in both visual fidelity and performance of our enhancements.

Chapter 7 deals with general implementation issues and system integration. Hardware, and Software considerations, as well as possible choices for graphics and programming languages are discussed. We introduce our object-model and explain how this integrates with the greater virtual reality system of which it is part.

Chapter 8 concludes our thesis and summarises both the problems addressed as well as novel solutions and contributions to each of our renderers.

## **1.6 Summary**

In this Chapter, we defined the problem statement forming the basis of this thesis; listed important milestones in general, non-NPR computer graphics; motivated the usefulness of individual NPR styles and introduced the document structure.

## 2 Related Work

In this Chapter, we examine the current state of the art in NPR rendering. We divide our review of related NPR work into the most commonly found NPR styles, but also consider the few publications that do not fit neatly into our categorisation. Some general but related technical terms are introduced, as well as general techniques, which help with our NPR rendering, but could also be used in various other graphical applications. We end this Chapter with an introduction to our performance test set-up, where we describe the Hardware and Software that our tests are performed on, the objects that are rendered with our NPR renderers as well as a default renderer, which is used for comparative analysis of our system's performance.

### 2.1 Non-photorealistic Techniques

We notice with pleasure that interest in NPR work has picked up over the last few years. Undoubtedly this is due partly to developments in the hardware sector that make customised rendering that much more viable and affordable. Various individuals and groups have dedicated their professional interest in creating non-photorealistic images. Most of the older work understandably is non-real-time, while recently several papers have been published and efforts been undertaken to perform NPR rendering at interactive frame rates. Examples of this work shall be introduced in the following Sections.

#### 2.1.1 General

One very exciting paper by Luebcke [46] addresses basically all known and common NPR renderers in one uniform and highly optimised framework. In the introductory Chapter, the author states that human artwork typically consists of a multitude of separate strokes. The techniques and tools to apply these strokes are what distinguish one artistic style from the next. The logical consequence for the author is to place view-dependent particles on the surface of a given object mesh (much like Meiers approach [51]). A multi-resolution mesh is used to speed up rendering performance and create local regions of increased or decreased spatial detail (for example near the viewer or on the silhouette). The unified rendering-framework then works in four steps:

- 1) Particle adjustment: the mesh-structure and correlated particles are adjusted depending on view-parameters and determine the stroke-density in the final image.
- 2) Optional Polygon Rendering: For some effects (like flat comic shading or depth-buffering) a solid-shape version of the model has to be rendered in one or more buffers
- 3) Transformation and lighting of particles: The OpenGL feedback mode is used to transform and light strokes (as well as obtain other information like local normal vectors etc. if necessary) into screen-space. Depending on the host-platform, this process might be fully hardware accelerated (the authors of [48] follow a similar object-space/screen-space hybrid approach, but never mention the possible use of hardware acceleration)

- 4) The output of stage 3 is used to place strokes in a user-defined manner. In fact, almost all steps involve some kind of user-callback function that determines the visual appearance of the final result.

Frame-rates achieved vary between 10-30 frames per second on an SGI Onyx with InfiniteReality Graphics. While the performance could definitely be higher for specialised rendering routines it is the great contribution of this paper to design a partially hardware accelerated framework that enables the user via a small set of user-defined callback functions to achieve any style from pen-and-ink drawings, to cartoon shading and even painterly strokes. No other paper we found was able to achieve such a diversity of styles in such a consistent and efficient framework. In addition to this, the important aspects of real-time performance and animation coherence were successfully addressed.

### 2.1.2 Comic Style

Our first exposure to comic rendering was in the form of a tutorial demonstration by Lander [43] who uses the technique described in Section 3.4. He renders objects with thick outlines and flat or extremely banded shading, producing an inexpensive but very convincing comic style look.

Lake et al [42], while demonstrating their work in an application using plugins (called *Inkers*) use very much the same technique.

A notable deviation from the prevailing theme is offered by NVIDIA [86], who, with the introduction of their latest GeForce class of hardware accelerated graphics adaptors, feature a mechanism called *Vertex Programs* [39] (Microsoft refers to similar features in their DirectX 8.0 release as *Vertex Shaders*, which NVIDIA argues is misleading, as shaders usually work on a fragment level and not a vertex level). These basically allow the programmer of a graphics system to write specialised rendering-code, which is then compiled into the native machine language of the Graphics Processing Unit (GPU) and uploaded to its working memory. Examples of this very powerful technique include a comic style renderer. By employing a normalising cube map (for a short introduction to mapping techniques, see Section 10.2.1), which is oriented relative to the viewer and white on the front half and black on the back half, the authors are able to produce a silhouette of varying width (a non-trivial task, as DirectX, the implementation language chosen by the authors, does not support the corresponding `glLineWidth` command of OpenGL). Other special DirectX constructs (special per-pixel normal dot products) in combination with alpha-testing and multi-pass rendering allow the authors to produce comic style images which are 100% hardware accelerated under DirectX on a GeForce card. (Microsoft [52] publish a very similar demonstration to showcase their latest DirectX release and it appears that in fact this work is achieved in collaboration with NVIDIA, so that we will not repeat the details here).

Markosian et al present a paper [48] which extends their original paper on the topic [41] and produces images of a very distinct look (in the original paper they tried imitating the style of Dr. Seuss [22]). They introduce new graphical units called *graftals* and *tufts*. Graftals are stroke-based procedural textures (in practice these are not textures in the technical sense, but filled triangle-strips with dark outline) that can

change shape and appearance to accommodate level of detail considerations. Tufts are structures that determine the multiresolution behaviour of a group of graftals. While the original paper mostly deals with placing and orienting the graftals in the scene, there were major draw-backs, namely temporal incoherence and visual popping resulting from graftals appearing and disappearing in the scene as the viewpoints changes. In the more recent paper, the authors address temporal coherence by fixing graftals to object-geometry (similar to Meier's [51] approach, but placing is performed at least partly manually, not unlike Cohen et al's [9]). Popping is avoided via a mixture of distance and time-based blending and level of detail functions. It is surprising that even though interaction is desired and problematic, all graftals are rendered with lines (usually with anti-aliasing – an extremely expensive operation and seldom hardware-accelerated for lines) and nothing is mentioned about the possibility of using billboards and texture-maps. The visual output is nonetheless very artistic and close to the initially attempted style.

Hall [28] developed a method which incorporates features of both comic style rendering and hatching, so that we will discuss it more closely in Section 2.1.3.

There are other papers, like Opalach et al. [63] (deals with squash and stretch effects that are commonly found in animated cartoon strips), Corrêa et al. [14] (maps hand-drawn textures on simple geometry for sophisticated cel animation), Petrovic et al. [67] (generates semi-automatic shadows from hand-painted scenes and simple geometric information for cel animation) or ten Hagen et al. [95] (performs facial animation on cartoon characters to limit the effects of imperfect facial expression reproduction), which deal with a large variety of related topics, but do not contribute to the actual rendering process of cartoon-style NPR.

### 2.1.3 Sketching

A variety of people have produced sketch style renderers and apart from the universal real-time/non-real-time distinction, there is another classification that can be made:

- Silhouette Rendering
- (Cross-) Hatching

While the first style is concerned mainly with the outlining of the silhouette (and/or important features, see Section 2.2.3 for details), the second style attempts to apply relatively flat shading (very similar to the standard comic style, see 3.1) using simulated ink or pencil strokes. While various examples of both styles exist it is interesting to see that very little work has been done in combining the two.

Lake et al [42] use projective texturing with four different textures to implement their pencil stroke (hatching style) *inker*. As textures can only be applied to complete geometric primitives (as opposed to parts thereof), they have to implement a geometry sub-division algorithm, to align primitive boundaries with shading boundaries. Unfortunately this process is processor intensive and not well suited for detailed objects.



Raskar and Cohen [70] use “*fattening of lines*” (extending the back-facing polygon of an edge outwards) and limited texturing to produce what they call “*Image Precision Silhouette Edges*”. While their method could more efficiently be implemented using texturing and thick lines (such as can be produced with OpenGL) they take great care to avoid resolution problems associated with depth-buffers of limited bit-depth through processes called z-scaling and line-fattening. Even though edges are rendered smoothly and precisely, they lack the sketched feel usually aimed for by this style.

Raskar later developed an extension of this approach in [69], where he shows how a standard rendering pipeline can be modified to incorporate the generation of Silhouettes, Ridges, Valleys and Intersections. As these geometric elements are used for a multitude of NP renderers it would be of great benefit to see them implemented in hardware. Another great advantage of his approach is that no connectivity information is needed between polygons so that they can be submitted to the graphics pipeline in a random fashion. He achieves this by extending each submitted polygon outwards by attaching segments at specific angles (the angles determine whether the extension creates a Silhouette, Ridge, etc.). The depth-buffer then takes care to eliminate the unwanted extensions in the interior of the object.

As mentioned above, Hall [28] presents a technique which he calls “Comic-strip rendering”, but which we categorise as a primarily hatching style. It involves a two-stage rendering scheme (intended for use with raytracing applications, i.e. non real-time), in which the first part uses a well-established lighting formula (e.g. Phong) to produce lighting information. A second stage then uses this information along with three-dimensional, procedural textures (in fact, their method is slightly more general) to produce grid-lines on the surfaces of objects. In his paper, Hall deals with the problem of matching up grid widths of different strengths and mentions anti-aliasing artefacts that will have to be dealt with. An interesting deviation from most other sketching implementations is that his hatching lines follow the object shape. This can be both desirable (it conveys three-dimensional shape information more clearly) and undesirable (object contours that are drawn too exact will look unconvincing and computer-generated).

Markosian et al [47] introduce a system which can render in a variety of outlining styles (exact – following object geometry, randomly perturbed and with texture attributes, hinting at pencil or chalk lines). Their main innovation is a very rapid edge detection, which uses three key factors: probabilistic identification of edges, inter-frame coherence of silhouette edges and fast visibility determination using a modified version of Appel’s hidden line algorithm [41] (for details see Section 2.2.3). The fact that they perform visibility determination in software (i.e. not using a hardware accelerated z-buffer) and still achieve interactive frame-rates for objects of considerable polygon-count shows the efficiency of their algorithm. Of all the sketching systems reviewed, theirs is the most flexible (with respect to sketching styles implemented) and most impressive (with respect to the visual output produced).

Another very realistic system was devised by Sousa et al [88], [89]. Their work achieves the realistic simulation of a variety of interrelated tools and techniques. Specifically they are able to model pencils with different hardness and surface-points, paper of varying texture and softness and the interaction between these two media. Furthermore, they develop a blender and eraser tool, both of which find regular



usage in the manual crafting of pencil drawings. The input to their system is highly pre-processed in that they assume information about visible edges, faces and shadows in normalised coordinate space readily available. This explains why the topic of hidden line removal is not mentioned. With this in mind it seems that their claim of interactive frame-rates is somewhat far-fetched (one example states rendering time between 7-10 seconds for a 100 face, 300 edge object). Still, their visual results are very convincing and the possibility of, for example, specifying a 2B pencil to draw on a semi-rough, medium-weight paper, is compelling.

Rössl and Kobbelt [73] use a three-stage system, which works on triangulated meshes to produce line-art drawings in a technical style. The use of triangulated meshes allows them to apply their algorithm to a large number of input objects and to obtain the necessary geometric data for their first (of three) algorithm steps. In this step several properties (including vertex, normal, shading value, local surface derivative, etc.) are stored in an *enhanced frame buffer* (similar to the *G-buffer* introduced by Saito et al. [77]) for later reference. The second step segments the original object into regions with homogenous principal direction fields. Some user-interaction is usually necessary in this step. The purpose of this segmentation is to identify areas, where a certain *style* (determined by the use of surface-lines and hatching-lines amongst others) is to be applied. The third and last phase then performs the actual tone mapping by using information gathered in the previous two steps to render quasi-parallel hatches of appropriate width onto the surface of the object. The authors see their system as a tool to help convert 3D models into line-art drawings and not so much as a fully automatic NP rendering system. Therefore they do not mention performance issues. The visual output of their work is suitable for technical illustrations, but seems rather limited in its stylistic capabilities (strokes seem too regular and computer-generated).

Another rather technical approach by the same authors [74] is mostly concerned with surface topologies and analysis of curvature gradients on these surfaces. Various parameterisations of triangulated meshes are introduced so that line-strokes can move along lines of greatest curvature. Seed-points are then distributed evenly over the surface of the object and grown into strokes from there based certain rules, determining stroke-density and with that shading, form, etc. The resulting images again look too computer-generated and resemble magnetic or electro-static field-lines more than a conventional line-art rendering. It seems that artistic merit and the element of randomness associated with so many NPR systems were sacrificed for mathematical discussion.

One of the few existing real-time systems is both artistically pleasing as well as partially hardware accelerated. Praun et al [68] use tonal art maps, which are intimately related to mip-maps (and are implemented through these), to place artistic strokes (dots, single lines or hatches) on the surface of 3D models. This is done in object-space (as opposed to the image-space approach followed by Lake et al [42] and ourselves) to follow isoparameter curves on the surface of objects. The tonal art maps allow the rendering to be coherent in regard to the distance to the viewer (strokes do not contract or expand as the object moves away from or towards the object) as well as in a temporal sense (strokes are fixed on the surface of the object and therefore avoid the notorious flickering or shower-door effects). Furthermore, art maps of lighter shading (fewer strokes) are subsets of darker maps (more strokes), i.e. darker maps are

generated by adding strokes to lighter maps. This facilitates the blending of various art maps to create a desired tone or shade. In order to vary the hatching density and style across a single surface, Praun et al use blending of 6 images (12 if considering multiple mip-mapping levels), but show several ways of how to optimise this using modern graphics hardware. The result is visually extremely pleasant and can even attempt to simulate textural properties such as short fur or the grooves and patterns of a human hand if the art maps are chosen and aligned carefully. All of this can be achieved in real-time on a powerful PC with modern graphics card (20-40 fps on PIII 933 with GeForce2).

Another real-time system (between 2 and 10 frames per second on high-end sun workstations for medium complex to simple objects) is proposed by Northrup and Markosian [59]. They use a hybrid algorithm that works in both object and image space. Firstly, the silhouette of an object is detected in object-space as described in earlier work by Markosian et al [47]. Then, the silhouette is projected into image-space and a series of operations and decisions is performed to generate long, continuous curves from the polygonal fragments comprising the visible silhouette in image space. The motivation for this is to more naturally replicate an artists fluid motion as she moves over the drawing surface and to represent an outline with long, determined strokes instead of short and angled ones. The last step is to render the image space silhouette curves with a variety of different styles. In order to allow for as large a variety as possible, the authors suggest any combination of the following stroke modifiers: antialiasing, tapering, flaring, wiggling, alpha fading and texture-mapping. The result here is not only very convincing (in the sense that the produced images appear hand-drawn), but also allows simulating a great palette of drawing styles and media (pen-and-ink, watercolour, pencil, etc.).

Deussen et al show a rather specialised use of a Pencil-and-Ink technique in [18]. Their main goal is to render trees (but shrubs and bushes can also be produced) in a distinct NPR style. With this in mind, they devise a system that is able to abstract from the full geometric detail of a given model while retaining the characteristics of the specific plant underlying the model. Stems are made up of cylindrical shapes and cross-hatched to indicate shape and texture. Foliage is modelled either by user-facing discs or a series of leaf-shaped polygons representing leaves from different views. Abstraction takes place automatically as a function of distance to the viewer, but can also be user-adjusted. Altogether Deussen et al. produce a system, which is capable of truthfully reproducing the style and feel of technically illustrated trees and other plants for architectural and landscaping purposes. Temporal coherence for animation as well as real-time performance were of no concern in this paper (even though a frame-rate of about three was said to be achieved for a low-resolution model on a high-end graphics station set to low graphics quality – improvements pending)

Schlechtweg and Strothotte [80] contribute to the field by attempting a line-definition that combines both artistic and mathematical backgrounds. They investigate the use of lines in line-drawings (purposes may be outlining, shading, but also to symbolise transparency and other abstract metaphors) and devise a set of measures that can be used to allot a certain amount of *drawing resources* to a specific object in a scene. These measures include: overall line count, overall line length, object visibility and others. The deliberate allocation of these measures can help to draw attention to parts of the scene or individual objects or to

limit the overall use of drawing resources (e.g. to adhere to performance constraints). Even though their main contribution is the definition of measures for drawing resources and the development of a user-driven editor to allocate the resources, the produced images have a certain artistic appeal (style: technical illustration).

In a paper by Salisbury et al [78], the authors use an augmented 2D image in conjunction with stroke textures to simulate technical illustrations. The input to the system are a shade map indicating the desired tonal value of the resulting image, a vector-field determining stroke-direction and finally stroke textures that are placed in an iterative process onto the image until the pre-defined tonal values are achieved. In the iterative process, blurred versions of the strokes are subtracted from the base-image in order to perform a kind of error-minimisation. This system, while working exclusively on 2D images (this is considered a feature not a flaw) is almost identical to similar 3D systems like [51], [18], [73], [74], [68], and [23] all of which use vector-fields or textured particles fixed to object-geometry (which the vector-fields achieve) or both to implement their various approaches. Still, visual quality of the rendered images is excellent. Real-time results cannot be obtained with this method, due to its required user-interaction and the complexity of the iterative rendering steps.

In addition to the work mentioned above, there are papers like Cohen [10], which deal with the topic of sketching (in this case using two-dimensional input devices to *sketch* three-dimensional object shapes) but which are not concerned with the stylistic aspects of the rendering process and which are therefore not discussed here.

#### 2.1.4 Painting

Very little work has been done on real-time painting of three-dimensional objects. Most interest in this area is focussed on filtering of still images or video sequences. A variety of commercial applications are available to produce painterly styles such as Oil, Pastel, Watercolour, Pointillism or Impressionism, to name but a few. Since these applications are aimed at producing the highest possible quality of output, real-time constraints are rarely of concern so that still images are produced within several seconds, while animated sequences take up to several hours or days to produce. Nonetheless there are a few examples of real-time video filtering or non-real-time painterly renderers that should be mentioned.

Hertzman and Perlin [32], [33] developed a system, which uses brush strokes to modify video footage into a painterly style. They use a large coarse brush to render an initial sketch of an image and, where this coarse image differs significantly from the original, render detail with refined brushes following contour lines. Time coherence is addressed by merely painting over the last frame (as opposed to re-rendering the following frame entirely) and using motion-detection techniques to avoid flickering of brush strokes. A certain customisation of the output is possible through the use of weight-maps (which allow a certain filter to be applied to certain areas of the original image) and the parameters to the rendering process. A *real-time* version of the system has been produced by the authors and is stated to take less than 4 seconds per frame on a dual-processor Pentium II 500MHz machine (a rather relaxed usage of the term *real-time*, as

discussed in Section 1.3.2). Nonetheless they argue that for the specific use of painterly rendering a frame-rate which is too high can lead to flickering and a visual appearance that is *too real*. They state that best visual results are obtained at frame-rates between 10 and 15. Another interesting result is that audience acceptance of an installation featuring an interactive version of Hertzman's system was generally high and that "*participants seemed to immediately understand and accept the process*".

Meier [51] presents a system capable of painting various painterly styles. She places importance on two common effects in brush-based systems: The shower-door effect, which occurs when brushes are fixed in screen space and the rendered scene seems to slide beneath these brushes; and the excessive noise that is created by totally random brush positioning. It is important to her to avoid both of these effects. The way she achieves her goal is by special object representation. Objects are specified using parametric surfaces. Depending on the painting style these are tessellated and brushes are actually fixed to positions on the object's surface. The number of brushes is related to the relative area of a tessellation unit with respect to the total surface area. Brush attributes such as colour, size, orientation and position are derived from object geometry as well as a reference picture (this is a standard smooth shaded rendering of the image, providing information such as lighting, colour etc.) and are stored either within the brush or as separate reference pictures (one for each of colour, orientation, etc.). After the brush attributes have been compiled, the brushes are transformed into screen-space and then depth-sorted (with respect to the viewer). The painter's algorithm (actually referring to a hidden surface removal technique and not related to painterly rendering) is then used to render the brushes onto the screen. As the brushes are fixed to the object's surface, the shower-door effect is avoided as well as extreme flickering. Even though no statements are given about the computational cost and other performance issues, the number of brush strokes necessary to fill the entire scene, as well as the cost of generating reference pictures and the fact that particles are manually sorted leaves us to believe that his technique performs far from real-time.

In the special context of a Virtual Environment, Klein et al [40] present a very simple but interactive approach. They model a given environment (in their paper an example was a virtual gallery) by texture-mapping photographs of real rooms or other pictures as texture maps onto the walls and doors of their virtual rooms. To achieve an NPR look, they apply image-filters (the like of which can be found in most modern image-processing packages) to the textures as a pre-processing step and draw non-planar edges with thick lines. Their main contribution, according to the authors, is the concept of art-maps, which are basically mip-maps of the textures at different resolutions. Whereas normal mip-maps at different levels are blurred versions of previous levels, the approach here is to apply the same image-filter to a smaller version of the previous level. This ensures that texture detail like brush strokes or other stylistic elements do not change in size as the viewer's distance towards the rendered objects change. Altogether this paper offers no really new rendering techniques with respect to NPR, but rather shows how existing NPR image-filters can be used in virtual environments to achieve a novel user experience.

Again there are other papers like Curtis et al. [15] or Baxter et al. [4], which deal with specific aspects of the computer-aided painterly process (in these cases the visually and physically correct simulation of media like Watercolour and Painting Brushes respectively), but are not directly applicable when



rendering 3D scenes in NPR styles. This is due mostly to the fact that the described techniques are developed for use in interactive painting systems though artists and are computationally very expensive.

### 2.1.5 Others

The above-mentioned techniques cover the majority of existing NPR systems. Alternative systems are few and far in between. If this is due to the fact that the above techniques describe most of the artistic styles found in every-day life or that users can most readily identify with these techniques is not known. Nonetheless there are some noteworthy exceptions.

Cohen et al [9] present not so much a NPR renderer as more an NPR world generation tool. Their main drawing primitive is the Billboard (a textured quadrilateral, which is maximally exposed to a viewer) onto which the user can draw and thus excludes the NPR rendering of standard 3D objects. The system is gesture-driven and in that context-sensitive (the user can create a terrain by drawing on the ground, a sky by drawing onto an enclosing sphere etc.). As the system reproduces exactly what the user enters in terms of drawing strokes, the achieved style is mainly user-dependent. Their main contribution lies in the specific use of Billboards and connecting Billboards (called *Bridges*) to re-render a given scene from novel viewpoints.

One of the only works that focus on dots for rendering is [17] by Deussen et al. Their goal is to simulate as close as possible the manually created pieces of art called *Stipple Drawings*. They argue that these types of drawings are commonly used in technical and scientific illustrations and are by their very nature easy to reproduce and inexpensive to print. As their starting-point they take half-toning values (a related but distinctly different technique) to distribute an initial set of stipple points. Voronoi-based dot-relaxation is then used iteratively to settle the points on the image. This is important to spread the points in such a manner that no obvious patterns are visible while retaining the intended distribution to suggest form, shade and structure. While this work can help reduce the creation-time of realistic (i.e. very close to manually produced) stipple drawings from days or even weeks to hours, it is obviously not a real-time process and needs user-intervention at several stages.

Streit et al [93] use an adaptive halftoning technique to implement their NPR renderer. This work is an extension of earlier work by two of the authors on *Importance Driven Halftoning* [92], which allows the user to specify an importance function and the types of drawing primitives to be used. The importance function can be specified in terms of an importance map (e.g. a secondary image) and influences the way the halftoning is performed locally. In an NPR context this importance map can well be derived from geometric information of a 3D model. Surface derivatives, the z-buffer and shading-values are but a few possible importance maps. These, in connection with the user-defined drawing primitives then determine the output of this particular renderer. In effect the authors show how 2D images of 3D models can be half-toned in various ways by incorporating some underlying information of the 3D model into the process.

Gooch et al [26] and [25] are not primarily concerned with artistic stylistic reproduction, but with the illumination process of NPR technical illustrations. Consequently, they devise a modified lighting model based on known lighting models like Phong shading to ensure that silhouette edges are always clearly visible (and not obscured by shading). Furthermore, the discovery of artists, that a light hue-shift to shading helps imply shading without requiring a large dynamic range, is implemented. The paper produces formulae to perform the lighting calculations and suggests how OpenGL could be used with several light-sources to implement a compliant visual result.

Interestingly, a fair deal of research has investigated the methods and uses of another rendering mechanism, which is very much NPR in nature, but is not mentioned anywhere in the NPR literature. We are referring to Stereograms or SIRDs (Single Image Random Dot Stereograms), which are two dimensional images, which, when viewed with the proper eye-technique, can give the illusion of a three dimensional scene. This illusion is produced by the brain due to the same mechanism that allows humans to derive depth cues from the slightly different images that the eyes perceive of a given scene. Some webpages and various authors ([62], [108], [97], and [96] to name but a few) take interest in this topic. Even though an implementation for our purposes would be fairly easy, we refrain from it because (along with the authors) many people have great difficulty actually seeing these special types of images. Nonetheless we believe that the classification of SIRDs as NPR in the classical sense is adequate.

## **2.2 Technical Terms**

Apart from the defining terms for this thesis which are discussed in Section 1.3, there are other, more technical terms, which are not directly relevant to real-time NPR rendering in specific, but which are needed for our geometrical discussions. These are defined in the following Sections.

### **2.2.1 Object Description**

The objects we consider all have the same underlying geometrical information and appearance attributes. This in fact is one of the main considerations in developing our renderers: to be able to render one and the same object in a variety of distinct styles without the need for augmentation of object-data. The objects in our system are thus completely described by the following data:

- A list of all the vertices making up the object
- A list of triangles, comprised of the above vertices, defining object-primitives
- A colour value for each vertex
- A set of different textures
- A list of texture indices and texture vertices for each triangle

It should be noted that while this set of attributes completely describes an object in our system, only the first two are guaranteed to be defined for each object. Therefore default behaviour is defined and implemented for objects lacking colour information or texturing.



### 2.2.2 Standard Vectors

During the course of this thesis vectors will play an important role in describing algorithms and evaluating their complexity. We therefore would like to very briefly introduce the vectors that will be used extensively.

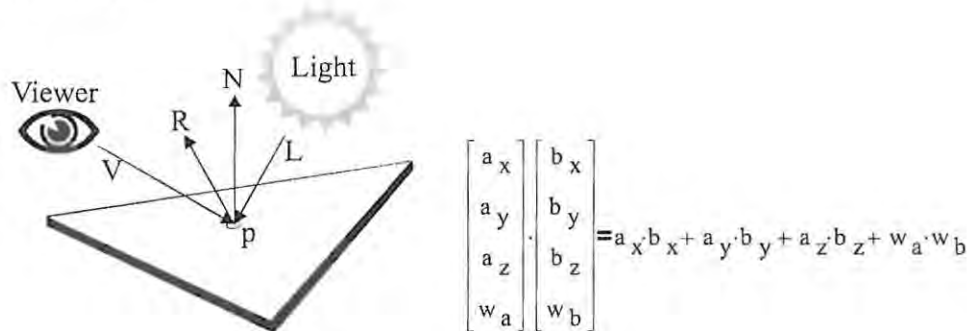


Figure 5 – Vectors: a) definitions; b) Dot product

Figure 5a) shows the following vectors:  $V$  describes the *view-vector* and is spanned between the viewer's position and the point  $p$  under inspection. The direction that the viewer is looking at, might be totally different and is irrelevant for our renderers.  $N$  represents a *normal-vector* (in this case of a triangle, but depending on the context it could also refer to the normal at a given vertex. Where the difference is important, we will denote the normal at a vertex with  $N_v$  and the normal of a triangle as  $N_T$ ).  $L$  indicates the direction of an imaginary *light-source*. We will make the simplifying assumption that the light source is infinitely far away and thus casting parallel rays (this assumption, while reducing computational overhead greatly, is not totally contrived, as the sun for example – our primary source of light - is far enough away from earth so that her rays can be considered parallel for most practical reasons). Finally, there is  $R$ , which is the *reflected light-vector*. This is to say that light, bouncing off a perfectly smooth surface with normal  $N$ , will reflect in the direction of  $R$ .

Figure 5b) shows the dot product of two four dimensional vectors (the fourth coordinate is called the homogenous coordinate). As we are using the dot product extensively to determine visibility and surface exposure for customised lighting calculations, we would like the reader to be familiar with the cost involved in performing such a computation. In most cases we perform the dot product on three-dimensional vectors, thus incurring 3 floating-point multiplications and 2 floating-point additions for each dot product.

### 2.2.3 Edges

Edges in many incarnations play an important role in a number of our renderers and so we shall provide a more refined meaning to the different nuances of these. In general an edge is found wherever two vertices are logically connected. Since a triangle is made up of three vertices, each pair of vertices makes up an edge. As vertices can be shared amongst adjacent triangles, so can the edges connecting these vertices be shared between triangles. If two such triangles lie in exactly the same plane (i.e. their normals are parallel), then we call the connecting edge an *inface edge*. Inface edges are usually not of visual interest

and so we filter them out when an object is loaded. In a well-defined and closed object each edge will represent the boundary of exactly two adjacent triangles. In our discussions we will consider only such edges. The edges that are of interest and their definitions follow:

*Silhouette or Outline* – these terms define a group of edges that possess the property that one of the adjacent triangles is facing towards the viewer, while the other one is facing away from the viewer.

*Significant edge* – this we define as an edge whose adjacent triangles exhibit a sharp angle. While in many cases they do not belong to the silhouette category, this is not implied but in most cases they represent object detail of considerable interest.

Various ways exist of determining the silhouette condition and we introduce some here. In Figure 6a) we show the physically correct method for perspective projected scenes: A vector is constructed from the viewer position to each of the triangles and dot products are computed between the normals of these triangles and the individual view-vectors. With this method triangle *b* will face the viewer, while triangle *a* will face away from the viewer. While this method produces the most accurate results, it involves re-evaluating the view-vector for each triangle – a large overhead.

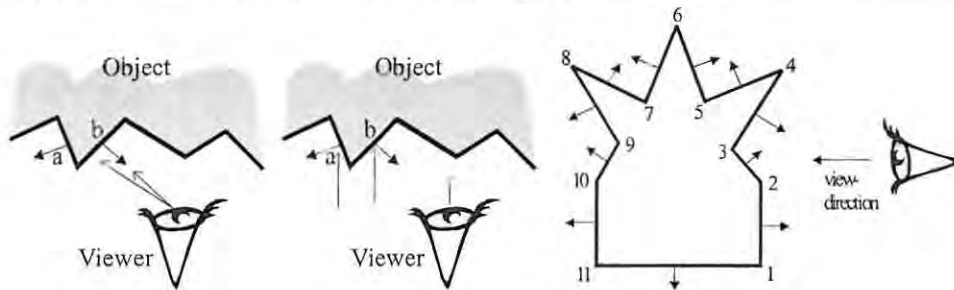


Figure 6 - Edge issues: a) Correct computation; b) Approximation; c) Artefacts

For objects of small dimensions compared to the distance to the viewer, we can make the approximation that all view-vectors will point in more or less the same direction (due to its reduced computational overhead, this is the method we use in our renderers). Figure 6b) shows how this approach will incorrectly report both triangles *a* and *b* as front-facing. It should be noted though that we constructed this example deliberately and that the viewer has to be placed extremely close to the object for the approximation to fail. Figure 6c) shows a slice-view through some object with the viewer looking from one side. Object lines thus represent triangles and numbers are aligned with edges (remember the slice-view). With this set-up edges 1,2,3,4,5,6 are on the front-side of the object (relative to the viewer), while the remaining edges are on the back-side. The approximation test will therefore identify edges 4,5,6,7 and 8 as silhouette edges, while 2,3,9 and 10 are normal edges. Edges 1 and 11 are not well defined, because the bottom triangle is neither front- nor back-facing. We'd like the reader to notice that silhouette edges can appear on the contour of an object (from a given view-point) but also anywhere on the front- or

backside of the object. The implication of this is that one has to deal with Hidden Line Removal where the need arises.

Microsoft [52] use another approximation which works well for fairly round and smooth objects. Instead of considering the normals of the triangles adjoining an edge, they look at the normals of the vertices making up an edge. They argue that if the angle between such a normal and the view-vector is (close to) 90 degrees, we are likely to deal with a silhouette edge. As the simplifying assumption can best be made of *round* objects with a high spatial resolution (and we use low-resolution objects for performance sake), we cannot easily make use of it.

Markosian et al [47] use yet another very innovative approach to rapidly detect edges, without the need for exhaustive searching. They argue that by starting on a known silhouette edge and tracing along adjacent edges, an almost complete silhouette can be found while testing only about 1% of the total number of edges. As they use a probabilistic approach, not all silhouette edges are guaranteed to be found, but long ones (which they argue are most important) have a proportionally high probability. They also state that the probability of an edge belonging to the silhouette is proportional to  $\pi - \theta$ , where  $\theta$  is the dihedral angle of an edge. Therefore sorting of edges according to probability will enhance the chances of finding the correct ones. The silhouette edges found in the previous frame are always checked in the next frame, thus exploiting inter-frame coherence. In addition to this, previous edges will be used as starting points for the previously mentioned silhouette edge-traversal. With all these optimisations in place Markosian et al claim a five-fold decrease in rendering time compared to the exhaustive search technique.

Sousa and Buchanan [90] describe an edge buffer construct to facilitate silhouette rendering. A hash table indexed by the lower vertex number of two connected vertices is used to store attributes such as *front-facing* or *back-facing* but it also extendible to hold user-flags. In practice we find that our own implementation is fairly similar. We use a linked-list instead of a hash-table as random indexing is not desired and therefore data-compactness more of a concern than look-up speed. Also, we store multiplied angles rather than just sign-bits, as this allows us to introduce other edges (which we call *significant* edges), which depend on the dihedral angle between adjacent triangles. Altogether, Sousa and Buchanan introduce a possible data-structure whose apparent advantages over other approaches are not immediately noticeable.

#### 2.2.4 Faces, Triangles and Polygons

The terms *face*, *triangle* and *polygon* all refer to primitives making up an object. In most cases they are used interchangeably. Three non-colinear vertices exactly define a triangle as well as a plane, which is why any triangle lies in exactly one plane. The same cannot be said for general polygons and these can therefore present mathematical problems as well as technical problems in the rendering stage, if they are not well behaved. For this reason all our objects are triangulated.

## 2.3 General Techniques

In this Chapter we would like to introduce several techniques which we have developed or refined and which are not specific to any particular rendering style. These techniques aide in the general construction of NPR scenes and in most cases can be used for a variety of rendering styles. The implementation examples we give are as always in OpenGL, but no specific constructs are used which would prevent a straight-forward conversion to another graphics language.

### 2.3.1 Custom Clear Operation

A basic generic rendering loop for a simple double-buffered display may look something like this:

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | ...); // clear all buffers
glPushMatrix(); // save initial matrix state
// render the scene
glPopMatrix();
glFinish(); // finish the scene
SwapBuffers(); // call some function that swaps the buffers
```

**Listing 1- Standard Render Loop**

The four component colour value (red, green, blue and alpha) will have been set by a previous call to `glClearColor(float, float, float, float)`. The result of the code in Listing 1 is an empty background, filled with a uniform colour, a uniform alpha value (if an alpha buffer is present) and a depth buffer that is reset to a value specified by the `glClearDepth(float)` command. To distract the consumer of our NPR renderers as much as possible from the fact that our images are rendered by a computer, we find it helpful to use non-uniform backgrounds (e.g. sketches appear much more convincing on paper, paintings are naturally at home on canvas). To implement this while retaining the highest level of performance, we designed the following code fragment to replace the standard clear operation:

```
glClear(GL_DEPTH_BUFFER_BIT | ...); // do not clear the color buffer
glCallList(backGroundImageList);
```

where `backGroundImageList` is a displaylist comprised of the following commands:

```
glNewList(backGroundImageList, GL_COMPILE);
glPushAttrib(GL_ENABLE_BIT|GL_POLYGON_MODE);
glColor3f(1.0,1.0,1.0); // full brightness on quad
glPolygonMode(GL_BACK, GL_FILL);
glShadeModel(GL_FLAT);
glDisable(GL_CULL_FACE);
glEnable(GL_TEXTURE_2D);
glDisable(GL_DEPTH_TEST);
glDisable(GL_LIGHTING);
glBindTexture ( GL_TEXTURE_2D, backGroundTexture); // get active
texture
// we want to make sure that the whole viewport is covered with this texture
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
// now that the matrix stack is saved and reset, we continue
glBegin(GL_QUADS);
glTexCoord2f(0.0,0.0);
glVertex2f(-1.0,-1.0);
```

```

        glTexCoord2f(0.0, REPEAT_FACTOR); // REPEAT_FACTOR determines
tiling of texture
        glVertex2f(-1.0, 1.0);
        glTexCoord2f(REPEAT_FACTOR, REPEAT_FACTOR);
        glVertex2f(1.0, 1.0);
        glTexCoord2f(REPEAT_FACTOR, 0.0);
        glVertex2f(1.0, -1.0);
        glEnd(); // quad
        glPopMatrix(); // restore ModelviewMatrix
        glMatrixMode(GL_PROJECTION);
        glPopMatrix(); // restore ProjectionMatrix
        glMatrixMode(GL_MODELVIEW); // Re-enable ModelviewMatrix
        glPopAttrib();
        glEndList();

```

### Listing 2 - Custom Clear Displaylist

At first glance this might seem like a lot of code to replace the `GL_COLOR_BUFFER_BIT` constant in the `glClear()` command, but it should be noted that this is the most general version of the necessary code if no assumptions can be made about the state of the rest of the system at the time of invocation of the displaylist. Lines 3 to 9 in Listing 2 for example are purely concerned with setting the OpenGL engine into the most efficient state for a full-screen texture mapping (McReynolds and Blythe [50], Node 266, argue that the number of state changes should be kept to a minimum, i.e. all scene elements requiring a certain OpenGL state should be rendered together. While this is certainly true, it should be contrasted with the performance loss of rendering a full-screen fill with unnecessary [depth testing] or even undesired [lighting] states enabled). Most of the remaining lines of code make sure that the correct matrix stack is activated and that the state of all matrix stacks is reconstructable. If assumptions can be made about the required state of the matrix stacks before and after the clear operation, we can delete these steps from the displaylist as well. In the general case they are necessary to ensure that the desired background image is guaranteed to fill the entire screen, independent of the orientation and position of the viewer or the dimensions of the viewport.

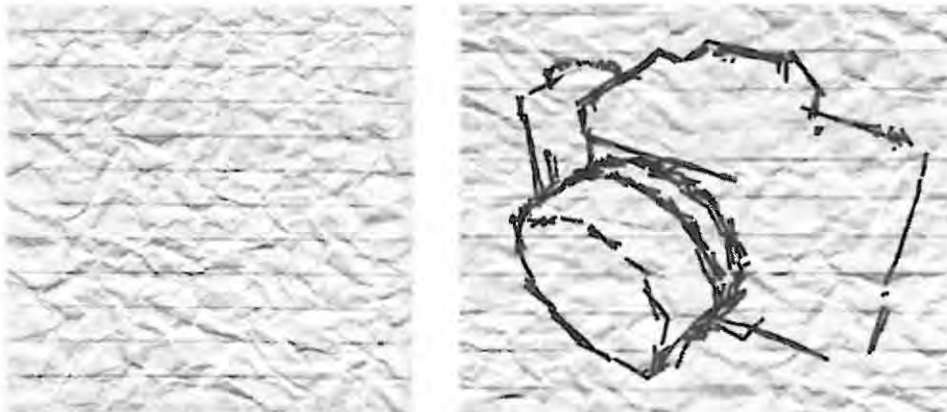


Figure 7 - Custom Background Clear:

a) tileable wrinkled paper texture; b) sketched camera on custom background

Figure 7a) shows a possible background texture. We created it by taking an actual piece of lined paper, crunching it up, unfolding it and scanning it in. After adjusting brightness, contrast and colour-distribution, we cropped the image into a square and applied a tiling filter to the image (i.e. the resulting image can be repeated horizontally or vertically without creating noticeable edges). Figure 7b) shows a



camera in random perturbation style rendered onto this background. For an animation of this technique, see Animation A (Section 9.2).

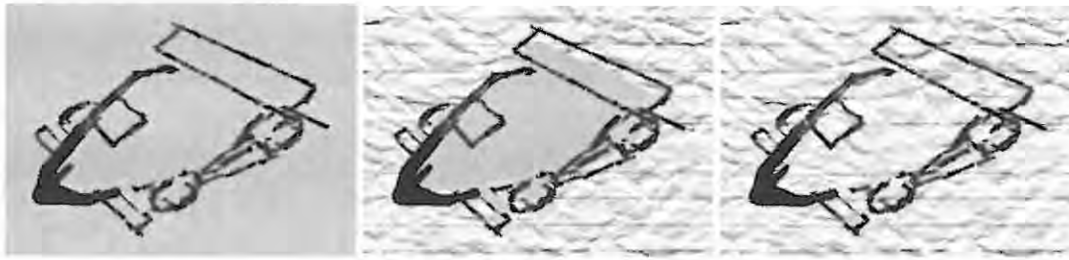
### 2.3.2 Hidden Line Removal with Background preservation

As Figure 7b) of the previous Section convincingly demonstrates, the presence of a non-uniform textured background can greatly increase the NPR appearance of a certain renderer. A problem arises when certain standard hidden line removal (HLR) techniques are used in connection with non-uniform backgrounds. HLR is a well-understood topic and many solutions to the problem have been provided. One of the most efficient ones is to use a z-buffer (if such is available). The basic algorithm for this approach is as follows:

- Render object in line mode in a colour different from the background colour
- Render object in fill mode in same colour as the background

This means that first all object lines are drawn (even those which are intended to be hidden). The depth values for these lines will be written into the z-buffer. Next, the object is redrawn, but this time using solid (filled) geometry. The reason this is done as a second step is to allow the depth buffer to reject such parts of filled segments, where non-hidden lines are already present (if we reversed the order, the object lines would be rejected by the z-buffer test). The depth value of hidden lines will be greater than those of filled fragments on the front of objects and therefore be overwritten. The background colour is chosen to fill the fragments, because this way they will be indistinguishable from the background and hidden lines will therefore be successfully erased. Figure 8a) shows this approach working well for a scene with uniform background, but in Figure 8b), where a texture background is present, it fails. We therefore had to modify the standard approach slightly to accommodate for non-uniform backgrounds.

In graphics systems where the depth test function can be set, we can reverse the two HLR steps. This is possible by defining the test function to be `GL_LEQUAL` (instead of the standard `GL_LESS`), which will pass for fragments that have a depth value which is not only strictly smaller than the one in the z-buffer, but for equal ones as well. This means that we can effectively initialise the depth buffer even before a line-object is rendered onto the screen. This means that instead of erasing unwanted lines, we prevent them from being rendered in the first place. The next trick is to prevent the colour buffer from being modified while the depth-buffer initialisation is taking place. This can easily be done in OpenGL, as there exist write-masks for all important buffers. The command `glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);` will prevent any of the r,g,b and alpha components from being modified. With this technique we are able to generate Figure 8c), a toy racing car with hidden lines removed and background intact.



**Figure 8 – HLR: a) standard approach; b) if background is present; c) with colour-masking**

It should be noted that with this technique the order of rendering objects becomes important. If distant objects are to be visible through near ones, they have to be rendered first (i.e. all objects have to be depth sorted according to the current viewpoint and rendered farthest to nearest). If distant objects are to be hidden by nearer objects, the situation becomes more involved. The initialisation of the z-buffer then has to be decoupled from the line rendering. This is to say that first all HLR objects have to be rendered into the z-buffer (the order here is unimportant, as the depth test will perform an inherent sorting), followed by rendering of all lines (the order here is likewise unimportant, as the correct z-buffer values will already be in place).

The HLR technique described here is not the most accurate one for all rendering styles, as we will discuss next. The most efficient way of rendering the above-mentioned HLR object into the z-buffer is via a displaylist. This implies that the geometry of the object is static and does not change. Two of our sketch renderers though, perturb the true geometry of objects to generate a sketchier look or to orientate strokes towards the viewer. This means that the geometry of the lines and the geometry of the HLR object are not identical (as is easily be verified by Figure 8b), where the HLR object is rendered in blue and slightly smaller than the line object in order to preserve important lines). Markosian et al [47] take this as grounds to implement a much more expensive, but accurate hidden line removal, which does not make use of a z-buffer. In practice (and Figure 8c) is a good example for this), the resulting artefacts (e.g. the front left spoiler) are barely noticeable and can in most cases be attributed to the deliberate imperfectionism imitated by the given renderer. This is demonstrated in Animation A.

## 2.4 Performance Testing

In Chapters 3,4 and 5 we perform a variety of qualitative and quantitative tests. The qualitative tests aim at demonstrating our techniques and design-choices and are different for each renderer. The quantitative tests are standardised in order to allow performance comparisons between the different renderers. We will therefore spend some time in explaining the general test set-up. The objects that are used for the testing, with all the necessary detail, are introduced next, so that specific information is only re-iterated in the individual Chapters where necessary. Finally, we will explain our definition of a *default renderer*, which is considered the fastest possible renderer that is still capable of reproducing all given object-detail (geometry, colour and texture).

### 2.4.1 Set-up

For all performance tests, we fix the following rendering variables in order to eliminate their potential effect on the results:

- Hardware & Software (see Table 1)
- Window-size (620x500=310000 pixels)
- Objects (see Section 2.4.2)
- Position of Viewer and Object (see Table 2)
- Orientation of Viewer and Object (see Table 2)
- Any OpenGL settings outside the rendering loop of a given renderer (all renderers restore 100% of affected state settings)

Thus only the difference between the various objects has an effect on the performance of the renderers.

Results are obtained by averaging performance values (in frames per second = fps) over at least 1000 frames for the comic and default renderers, at least 500 frames for the sketch renderers and at least 200 frames for the painterly renderer.

Hardware	
CPU	Intel Pentium II @ 500Mhz
Memory	192 MB
Graphics Card	GeForce DDR: <ul style="list-style-type: none"> <li>• Clockspeed=140Mhz</li> <li>• Memspeed=360Mhz</li> <li>• Resolution: 1152 x 864 (32 Bit)</li> </ul>
Harddisk	Quantum Fireball lct10 15Gb
Software	
OS	Windows 98 SE
OpenGL	1.3
Graphics Driver	NVidia® for Creative Labs® Version 21.85

Table 1 - System Set-up: Hardware and Software

	Viewer	Object
Position (3D Point)	12.73, 17.85, 12.52	0.0, 0.0, 0.0
Orientation (Quaternion)	0.866, -0.091, 0.490, 0.051	1.0, 0.0, 0.0, 0.0

Table 2 - Test Set-up: Positions and Orientations

### 2.4.2 Objects

The objects used in our performance tests are level-of-detail variations of one and the same base object. The reason for this is to vary the number of triangles comprising the object, while keeping the following as constant as possible:

- Screen-footprint (percentage of screen-pixels occupied by object-pixels)
- Euler's Relation of Faces, Triangles and Edges [111]

While it is very difficult to keep all of these constant during a level-of detail reduction, we keep the variations within certain bounds (13%, see explanation below). The level-of detail reduction was performed in such a way as to approximately halve the number of faces each time, while keeping the above-mentioned ratios near constant. Table 3 lists the object-statistics that are relevant to most of the renderers. In the first column we simply state the object name, the second and third columns show the number of vertices and faces, respectively. Columns four and five show how many faces are front-facing and back-facing, respectively, and are view-dependent according to Table 2. It should also be noted that correct view-vector calculations per triangle were used to determine these numbers. In column six we list the total number of edges of the object, while column seven lists the view-dependent silhouette edges (again with correct perspective and no approximations). The eighth column shows how many of the total window-pixels are occupied by the object. Finally, the last column details the above-mentioned Euler-relation expressed as a ratio  $(V+F/(E+2))$ , relating the number of edges to the sum of the vertices and faces and we note, that this ratio keeps close to 1.0 with a range of 13% .

Name	Vertices	Faces	Front Faces	Back Faces	Edges	Silhouette	Screen Coverage (%)	Euler's Ratio
Deer0	350	578	299	279	814	182	14.08	1.14
Deer1	502	870	461	409	1250	248	15.20	1.10
Deer2	709	1284	632	652	1871	295	16.08	1.07
Deer3	1148	2162	1050	1112	3188	430	16.72	1.04
Deer4	2811	5488	2555	2933	8177	733	17.20	1.01
Deer5	5392	10648	4843	5805	15916	1098	17.26	1.01

Table 3 - Universally relevant Object Statistics

Figure 9 graphs 3 traces on a log-log scale. The legend reads "*Vertices vs. Edges*" for the first trace. This places *Vertices* on the y-Axis and *Edges* on the x-Axis. The other traces are to be read accordingly. From the straightness of the traces, we can deduce the near-constancy of the associated ratios.

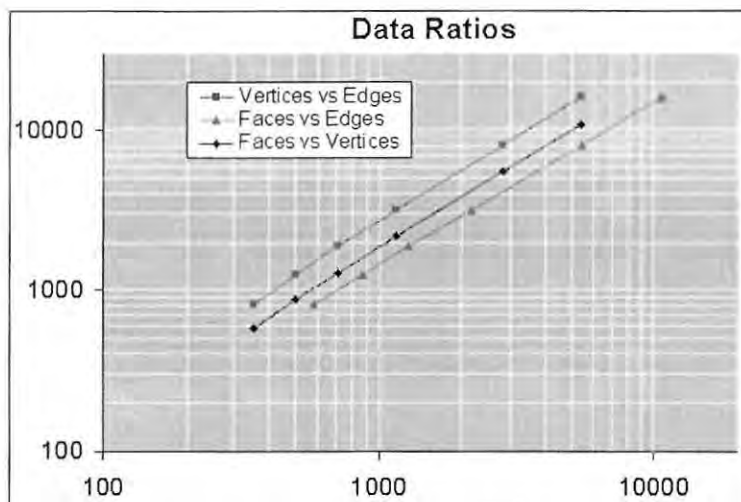
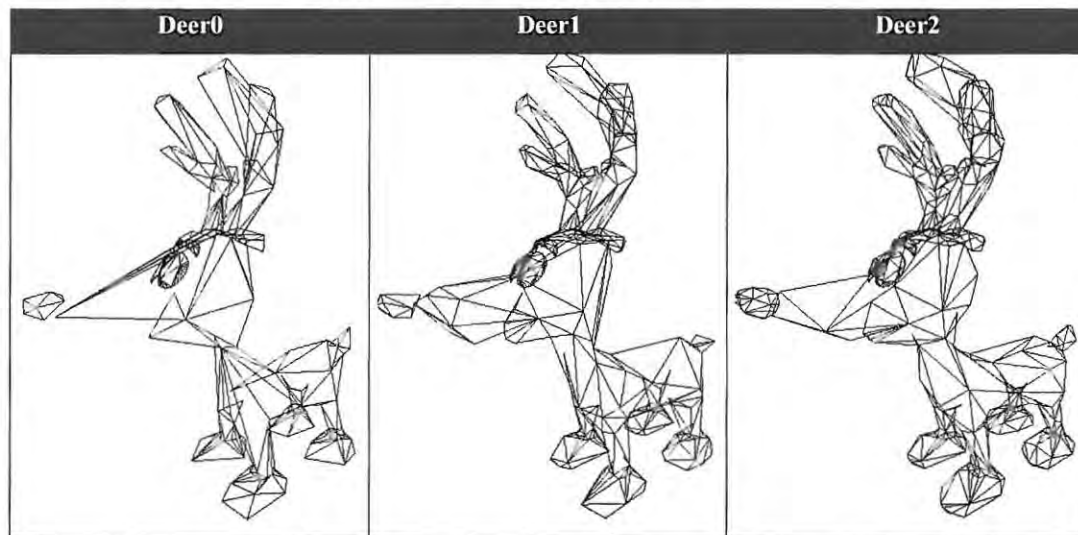


Figure 9 - Data Ratios for Test Objects

Table 4 shows how the different objects compare visually. The images demonstrate the resolution of the objects, as well as the relative positions and orientations of viewer and objects. It should be noted that the given images are cropped to fit the table and that the actual output window-dimensions are, as stated above, somewhat larger. We also would like to draw attention to the visual for Deer0. This image shows the nose of the reindeer severed from the rest of the body as a result of the level-of-detail reduction. While this was not necessarily intended, it reveals that we make few assumptions about well-behaved properties of our objects (i.e. our algorithms do not rely on objects being concave/convex or even continuous). Some renderers may produce artefacts for open objects.





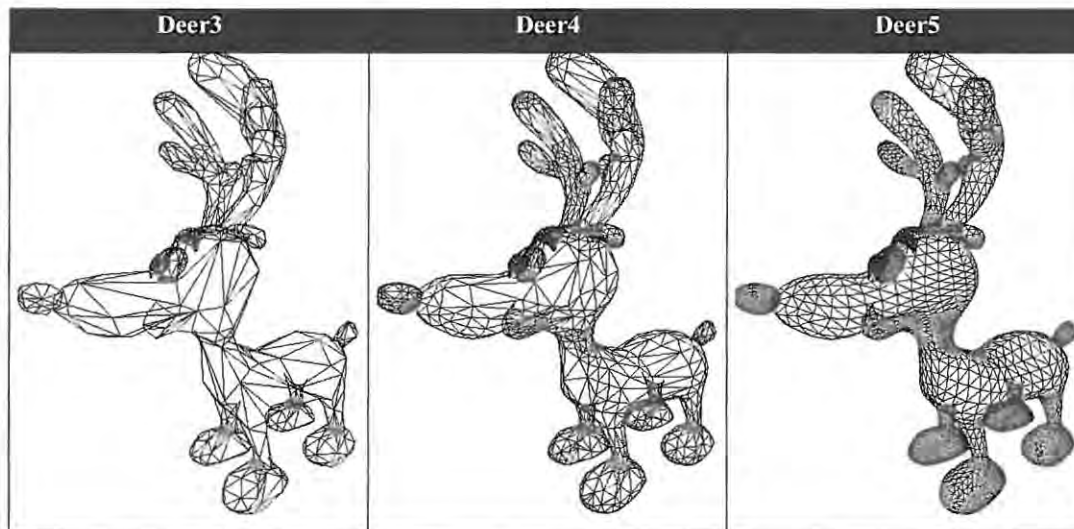


Table 4 - Visual Comparison of Objects

### 2.4.3 Default Renderer

In many situations, we reference a so-called *default renderer*. The performance of this renderer is defined as 100% and, where desired, our other renderers are measured against it. The following OpenGL settings (we only list non-default settings) are characteristic:

- Face Culling
- Depth Testing
- Lighting, 1 Light
- 2D Texturing, where applicable

For optimisation purposes and because the geometry of the objects do not change for the default renderer, objects are compiled into displaylists and thereafter rendered fully hardware-optimised in a single pass. In many cases we talk about both the *default* renderer and a *standard* renderer/approach and it is important not to confuse the two. The *default* renderer has been explained above. The *standard* renderer/approach is the basic or standard or customary implementation of any of our NPR renderers, before extensions or optimisation.

Figure 10 depicts typical examples of objects rendered in the default style. The diffuse lighting effect is clearly visible in Figure 10a), as is the smooth shading applied to the object. Figure 10b) demonstrates how texturing affects the rendering result.

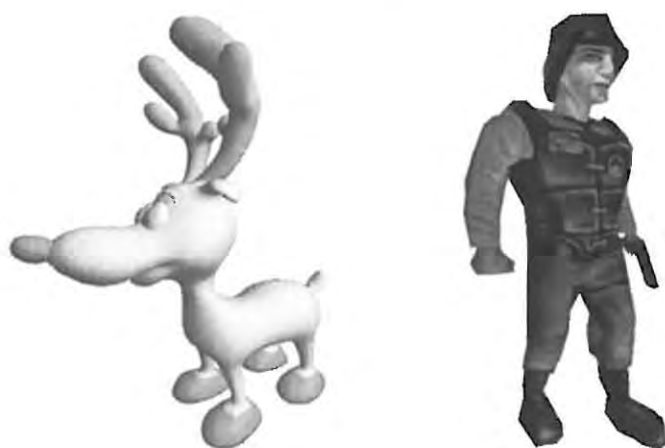


Figure 10 - Default Renderer Examples: a) Deer5; b) Barney

## 2.5 Summary

In this Chapter, we expanded on the current state of the art in NPR rendering. While we only found one paper which details a possible frame-work for combining the most common NPR styles in one orthogonal system, there exists a sizeable collection of work pertaining to individual styles.

Several comic rendering systems are introduced, most of which use the latest techniques in graphics hardware configurability to perform fully hardware-accelerated comic style rendering.

By far the greatest body of related research papers deals with NPR sketching. We identify the most common implementations of

- Silhouette sketching
- Hatch-style sketching

The strengths and weaknesses of the individual approaches are assessed and noted. In comparison to the comic style solutions, approaches to sketch rendering are considerably more diverse. Some authors work in object-space, while other work in screen-space. Some use perturbation of object geometry to simulate human imperfection while others use texturing. A few papers address physical media simulation like pencil-shape and softness. Some systems introduce algorithmic optimisations to achieve real-time performance.

A very limited number of papers deal with NPR painting and real-time performance is usually not a concern, but interesting results with respect to placement of brushes are obtained. Other related work investigates the physical modelling of realistic brushes and paints, but not necessarily in an NPR context and definitely not in real-time.

Most NPR related work can very neatly be classified as one of the main NPR styles of comic, sketching and painting, but a few interesting exceptions exist. We list some of these and discuss their relevance to our work.

Following this, we define some technical terms which are used in our geometrical approximations, optimisations and general discussions, but which are not directly related to NPR or real-time considerations.

Next, we describe some of our own techniques, which help us with our NPR rendering, but are not NPR in essence. The custom clear operation enriches our rendered scenes by supplying a custom backdrop instead of the standard uniform clear colour. A slightly modified hidden line removal technique allows us to hide unwanted object lines by using the z-buffer without affecting other scene elements.

Lastly, the set-up for our performance testing is introduced, which we use throughout this thesis to allow for comparative analysis of the results obtained for our individual NPR renderers.

In summary, we note with delight that NPR is gaining in popularity in recent years and as a result of this a number of interesting and relevant publications are available. Various ideas and techniques of existing work have helped us form a better understanding of NPR and were gainfully employed in many of our renderers.

## 3 Comic style

### 3.1 Introduction

#### 3.1.1 Definition



Figure 11 - Various Comic Styles [75]

Figure 11 shows a small glimpse of the large diversity of drawing styles that fall into the comic or cartoon category (the dictionary defines a comic as a series of cartoons, but we shall use both terms interchangeably) and due to this many authors define the comic rather by its purpose or ideology than its visual appearance. Nonetheless, many popular comics share some common characteristics, as shown in Figure 12. While the looks of possible characters are widely different and usually defining for a certain cartoon (*Futurama* for example uses big round eyes and protruding lips on all their human characters), it is the following common characteristics that tie most comics together:

- A thick dark outline of the silhouette
- Other thick dark lines detailing object characteristics
- Uniform or very banded shading

While the concept of comics is undoubtedly larger than the sum of the above-mentioned stylistic elements, we are interested in our work to create a rendering of a typical comic look and are thus satisfied to use schemata that most people will identify as comic-style.





Figure 12 - Typical Comic Style: A scene from Futurama [19]

It would go beyond the scope of this thesis to thoroughly analyse a representative amount of available comics with respect to the above characteristics, so we will proceed to examine in some detail the above (very common) look of Figure 12. We find that the silhouette is drawn approximately twice as thick as the thinnest lines (many comics in fact only use one stroke-width, but we find that it enhances the comic-appearance to use a thicker silhouette). Furthermore, we investigated the colour values of the different shading regions, the results of which are detailed in Table 5.







Detail	Description	Bright Value			Dark Value			Delta (%)		
		H	S	B	H	S	B	$\Delta H$	$\Delta S$	$\Delta B$
	Bender Body	191	28	90	191	34	74	0	6	-16
	Fry Hair	24	100	100	15	100	80	-2.5	0	-20
	Fry Jacket	337	100	80	353	100	57	4.44	0	-23
	Fry Pants	198	69	95	198	83	78	0	14	-17
	Leela Skin	348	25	100	344	30	91	-1.11	5	-9
	Scooter	51	100	100	45	100	92	-1.67	0	-8
		Average:						-0.14	4.17	-15.5

Table 5- Duo Shading Colour values for a given sample set

We have chosen to give the colour values in the HSB (Hue, Saturation, Brightness) colour space, because it is best suited to compare the dark and light regions of interest. By studying the table, we find that the artist has chosen to use highly saturated colours except for skin regions (Bender's (the robot) skin is in this case metallic), which have a paler appearance. In this colour space, the Hue and Saturation values very much determine the mood and atmosphere of the comic in the sense that diverse and highly saturated colours will convey a happy mood, while similar and earthy colour in low saturation will yield a gloomy, low-contrast look. In this case we are mostly interested in the light and dark regions of the two levels of shading so that hue is actually of little concern. We find that in this case, that many light regions have maximum brightness and dark regions are in average 15.5 percent darker. To establish whether these values are specific to the Futurama comic style or applicable in a wider sense, we examined 35 further samples from 12 different cartoons and found that brightness in dark regions decreases by  $(24.2 \pm 7)\%$ , while saturation increases by  $(12.0 \pm 9)\%$ . These large deviations can be explained by the considerable difference in style of our samples and it should be noted that variations within a given style are usually much smaller. In summary, we can say that saturation may slightly increase for darker areas, and brightness is usually decreased by a fair amount. We conclude from this, that a drop in brightness of about 10-30 percent, with other colour values staying constant, should result in a reasonable comic-shading look.

## 3.2 Problems

### 3.2.1 Problem Statement

As becomes evident from Section 3.1.1 we need to be able to solve the following problems in order to render objects in a recognisable comic style:

- Identify the silhouette
- Identify other important folds and creases in the object
- Render these features with thick dark lines
- Shade the surface of the object in a single colour or
- Apply a heavily banded shading to the surface

### 3.2.2 Implementation-specific Problems

Identifying the silhouette and other important geometric features may be computationally expensive and needs to be highly optimised to allow for interactive rendering behaviour.

There does not as yet exist a mode in OpenGL that allows the rendering of silhouettes or other specific object edge-data (Raskar [69] proposes a feasible hardware-accelerated approach that could easily be incorporated into OpenGL and other graphics APIs). One can render objects in line or wire-frame mode, but that will draw all lines, not solely those belonging to the silhouette.

Line-width in OpenGL can be specified with `glLineWidth()`, but need not be implemented in the GL-driver (as we find on some windows systems).

Uniform or flat shading can easily be accomplished by only having one global light-source providing an ambient lighting term and specifying `glShadeModel(GL_FLAT)`. In order to implement heavily banded (i.e. discontinuous) shading which is not limited to the boundaries of surface elements, we have to enable some kind of interpolation across surface elements. The alternate `glShadeModel` option (`GL_SMOOTH`), will interpolate colours smoothly using Gouraud shading and is therefore not adequate. A suitable interpolation scheme has to be found.

### 3.3 Solution

The solution we adopted for our system originates from Lander [43], and makes use of different rendering modes in a multi-pass rendering scheme to achieve its goal. The basic algorithm is shown in Listing 3:

```

Let L = Light Vector
Let Nx = Normal (at any of Vertex, Triangle, etc.)
Let View = View Vector
Let S[] = Lookup table of Banded Shading Values

Render
  // Perform lighting calculations per Vertex
  For each Vertex V do
    compute Nx.L
  Next Vertex

  // Render the silhouette
  For each Back-facing Triangle Tb do
    DrawWire Tb      // Draw the edges of Tb with thick dark lines
  Next Triangle

  // Render the interior
  For each Front-facing Triangle Tf do
    DrawFilled Tf     // use the lighting calculations from above
                      // to index S[] and colour the corners of Tf,
                      // interpolating them across the surface of Tf
  Next Triangle

End Render

```

**Listing 3 - Standard Comic algorithm**

The working of the algorithm is as follows: Light-Intensity values are computed for each vertex. As a rough but suitable approximation, these values are directly proportional to the dot product of the normal at the vertex with the light-vector. The dot product of two vectors is proportional to the cosine of the angle between them and therefore gives an indication of how directly a light vector hits a given surface. If the normal vector and the light vector are anti-parallel exposure is maximal and the dot product is  $-1$  (assuming the vectors are normalised). If the vectors are at right angles, the light passes parallel to the plane of a surface element and no light hits the element. The dot product in this case is 0. The reader should note that the dot product represents only the diffuse light component of the more complete Phong reflection model. While this suffices to apply comic style shading, we experiment with the specular component as well to achieve different visual effects (see Section 3.6.1 for details).

In the first rendering pass, all triangles are rendered in a wire-frame mode with thick dark lines. The notion of *thick* in this case means that the three edges that define a triangle (Figure 13a – two adjacent

triangles are shown here) will be rendered into the frame-buffer with lines that are several pixels wide (Figure 13b) and are intended to extend over the real triangle dimensions (Figure 13d) (this method is explicitly used by [70] and extended in [69] and inherently available in OpenGL)

The second rendering pass then renders over the existing image. The triangles are rendered in their exact geometric dimension in fill mode (Figure 13c). As this will eliminate all interior edges, we arrive at Figure 13e), where the diagonal is not visible anymore, but the silhouette is drawn in thick dark lines.

The major advantages of this approach are that we do not need to identify silhouette edges explicitly, it will work not only for the contour of the object but any edge that fulfils the edge-condition (as defined in Section 2.2.3) and that we do not need to be concerned about perspective projection artefacts, because the silhouette is generated in screen-space.

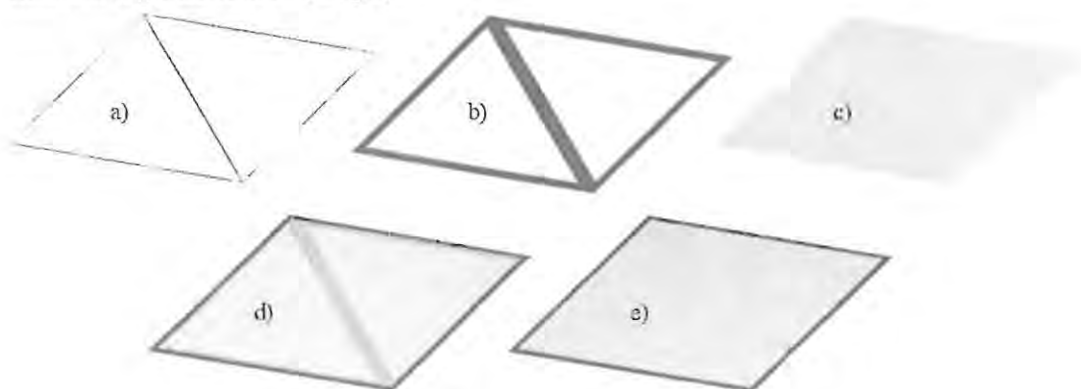


Figure 13 - Comic two-pass rendering principle

To accomplish the banded shading, we use a look-up table with colour values (for a single colour all entries will be equal, for two colours some percentage of the entries will have one value and the rest another, and so on – see Figure 22a) for an example). This makes it possible to easily specify the number of bands and the relative sizes of each. To fill a given triangle, the lighting-values of its three vertices are used to index into the look-up table. We fill the interior of the triangle by interpolating between the vertex lighting values and using the interpolated results to index into the table, thus allowing a shading boundary to run smoothly through a triangle. In other words, not the colours are interpolated across a surface (as in Gouraud shading), but the index-values. Since the number of distinguishable elements in the look-up table is limited and fixed, the number of colours applied to the surface element is similarly limited

### 3.4 Standard Approach

Lander [43] implements the algorithm of Listing 3 in a slightly varied fashion to improve performance by taking advantage of OpenGL specifics. Firstly, rendering front-faces and back-faces separately requires identifying them (doing this manually is a considerable overhead and requires perspective correction, as we discuss in Section 3.5.2.1). Instead, Lander first renders all triangles in fill-mode and instructs OpenGL to cull back-Faces (this is not strictly necessary, but will improve performance). He also makes use of the Depth-Buffer and initialises it with the first rendering pass. The look-up table discussed above is loaded into OpenGL as a one-dimensional texture. Indexing is performed by specifying a texture-

coordinate. Interpolation of colour-values across the surface of triangles is done automatically by the OpenGL texture-unit. The shading-texture thus modulates the original surface colour and appears as expected as banded shading.

In the second pass Lander again renders all triangles, but this time culls front-facing ones. Since no lighting calculations need to be performed for the silhouette, he can use a displaylist to greatly improve the performance of the second rendering pass. By rendering triangles in line-mode and by using the previously initialised Depth-Buffer, Lander can render the silhouette into the existing frame-buffer without disturbing the object interior (and actually rendering only a minimum of lines).

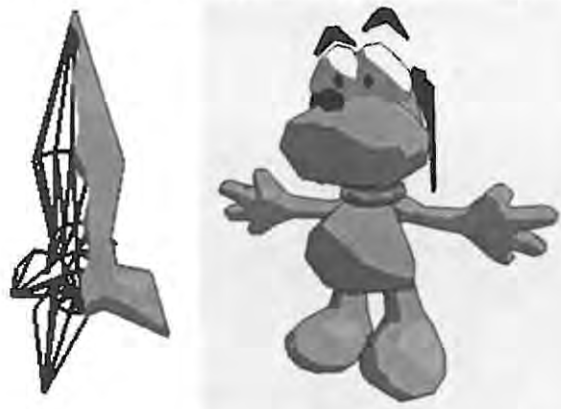


Figure 14 - Standard Comic rendering: a) Rocket; b) Dog

Figure 14 shows how this working in practice. Figure 14a) demonstrates the two rendering passes (the wire-frame of back faces on the left, and the filled front faces on the right). Figure 14b) illustrates how this technique allows us to generate a flat-shaded object with one shade colour and one highlight colour. To view a scene rendered with the standard comic renderer, see Animation B.

## 3.5 Optimisations

In the following Sections, we discuss optimisations to the standard comic renderer with respect to both performance and visual quality. The Section about Geometric Redundancy and the associated Perspective Correction are quite detailed, but can be applied to other areas of computer graphics, where face orientation has to be established quickly and with an adjustable amount of accuracy.

### 3.5.1 Geometric Redundancy

Even though Lander's implementation works well and is fast, we notice a large degree of geometric redundancy (due to the fact that the entire object is rendered twice). This does not matter much, because modern graphics cards can render a triangle in the same order of magnitude of time as it takes to determine approximate face-orientation. If we want to extend the lighting model, we have to perform additional calculations on each vertex and these calculations soon become the dominating factor in the rendering process. For this reason we decide to determine face-orientation with the following goal in mind: Even though face-orientation determination will introduce further load on the system, it would still benefit us if this load were to be balanced by the decrease of vertices that have to be included in the



lighting calculations as well as the decrease in triangles that have to be rendered. To allow for comparisons, we define the following relationship:

$$\text{Tri} = a \cdot \text{Ver}$$

This means that each object has a fixed ratio of vertices to triangles (Euler's relationship [111] describes this ratio including the number of edges for closed, convex, polyhedral objects – restrictions which we do not strictly enforce). For most of our test-objects it holds roughly that  $a=2$ , i.e. there are about twice as many triangles as there are vertices. While the exact number will vary from object to object, it is important to note that in general there are many more triangles than vertices.

By lacking information about face-orientation, we have to compute lighting values for each vertex and then render all triangles twice (once for the silhouette and once for the interior). The cost of the standard algorithm is thus:

$$\begin{aligned} \text{Ver} \{ \text{lighting values} \} + 2 \cdot \text{Tri} \{ \text{rendering front\&back} \} = \\ \text{Ver} + 2 \cdot a \cdot \text{Ver} = \text{Ver} \cdot (1 + 2 \cdot a) \end{aligned}$$

### 3.5.2 Face-Orientation Determination

There are various ways to determine whether a surface element is pointing towards the viewer or away from her. Some pointers on this topic have already been given in Section 2.2.3.

The correct way to establish face-orientation is to spawn a vector from the viewer's position to the centre of the face under question and calculate the dot product of this vector with the normal to the face. If this dot-product is negative, then the face is front-facing otherwise it is back-facing. This method is prohibitively expensive, because of the need to spawn a vector from the viewer to every surface element.

Our first approach was to use an approximation and assume the view-vector to be constant (resulting artefacts are discussed in the Section 3.5.2.1). We thus traverse all triangles of an object and mark each face either front or back-facing. While doing so, we also mark the vertices defining a front-facing triangle. Next, we traverse the vertex-list and compute the lighting values for marked vertices (i.e. for those that are part of front-facing triangles). Finally, we render only front-facing triangles.

If we let  $b$  be the proportion of front-facing triangles and  $(1-b)$  the proportion of back-facing triangles, we end up with the following cost:

$$\begin{aligned} \text{Tri} \{ \text{Face orientation} \} + b \cdot \text{Ver} \{ \text{lighting values} \} + b \cdot \text{Tri} \{ \text{rendering front} \} + \\ (1-b) \cdot \text{Tri} \{ \text{rendering back} \} = \\ 2 \cdot \text{Tri} + 0.5 \cdot \text{Ver} = 2 \cdot a \cdot \text{Ver} + 0.5 \cdot \text{Ver} = \text{Ver} \cdot (b + 2 \cdot a) \end{aligned}$$

We make the assumption that the percentage of front-facing triangles  $b$  is equal to, or very close to, the percentage of vertices that are part of front-facing triangles. We therefore reduce the cost compared to the standard approach as long as  $b < 1$ , i.e. as long as not all faces are front-facing.

Another, less obvious approach is to use the previous approach backwards. We see if a vertex is front or back-facing and mark the attached triangles accordingly. The reason why we would want to do that is

because in a well-behaved closed object, the number of vertices is much smaller than the number of triangles. We discover that there are other advantages to this approach, but we first need to explain, what the normal of a vertex is. As a vertex is a point in space it can, by geometric definition, not have an orientation. Nonetheless, the vertices under consideration are part of an object-structure and their normals are thus defined as: The normal of an object-vertex is the average of the normals of all triangles of which this vertex is a part. This is the reason why we can infer, to some extent, the orientation of a face from the orientation of one of its vertices. Obviously, the normal of a vertex is in most cases not equal to the normals of its attached triangles, but for finely tessellated objects the difference is small. This is important, because we are more concerned about speeding up rendering for objects with many faces, as smaller ones will render faster anyway.

Our algorithm was then as follows: Traverse the vertex-list and determine the orientation of a vertex. If it is front-facing, perform the necessary lighting calculations and mark the attached triangles as front-facing. Finally, render only front-facing triangles.

The drawback of this approach is that many triangles can be attached to one vertex (the inverse relation is much more well-defined, as a triangle can only ever consist of exactly three vertices) and so a lot of time is spent on marking the same triangles as front-facing. Knowing the ratio of faces to vertices of a given object, we tried to use a statistical approach in which only a certain percentage of attached triangles are marked in the hope that other triangles will be marked by neighbouring vertices, but unfortunately this can leave constant holes when an object-synchronised approach is taken. Using a randomised approach leaves less holes, but produces *clicking* (sudden appearance and disappearance of holes). Another problem is that triangles can be set front or back-facing by several vertices so that in a border situation we can get the same triangle set front-facing by one of its vertices and back-facing by another, so that the last processed vertex will dominate. This can obviously result in holes in the object and solving the situation involves either a kind of majority approach, re-setting face-orientation before each rendering pass to an undefined value or using what we call a *large flag* instead of a standard binary flag. None of these solutions is really optimal, as the issue of accessing many triangles per vertex still remains.

We therefore abandoned the idea of marking triangles in the vertex-traversal. Rather, we check the vertices, when triangles are rendered. This has the added advantage that different rules can be applied to the rendering phase that will result in varied visual quality and performance. The rules we investigated are: All three vertices of a triangle have to be front-facing for the triangle to be considered front-facing (this is the most restrictive condition and will render the least triangles, resulting in the best performance and the worst visual quality); at least two vertices have to be front-facing and finally, only one vertex has to be front-facing. The rule which is chosen can be changed dynamically at run-time and made dependent on the distance of the viewer to the object as well as the maximum deviation of a vertex normal from its attached triangle normals (which can be established upon loading the object).

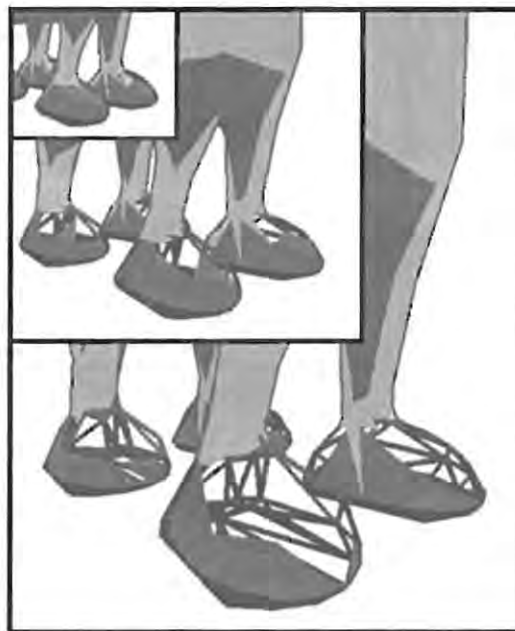
The updated cost is therefore:

$$\begin{aligned} & \text{Ver} \{ \text{Orientation Determination incl. lighting} \} + b \cdot \text{Tri} \{ \text{rendering front} \} + \\ & (1-b) \cdot \text{Tri} \{ \text{rendering back} \} = \\ & \text{Ver} + \text{Tri} = \text{Ver} + a \cdot \text{Ver} = \text{Ver} \cdot (1+a) \end{aligned}$$

Using our new approach to face-orientation determination we are able to achieve a significant increase in rendering performance and introduce a level-of-detail measure that allows for dynamic run-time adjustment of rendering quality and performance. It should be noted at this stage that the above cost calculations will be slightly skewed when implemented. This is due to the fact that part of the algorithm will run on the host computer (e.g. custom lighting calculations and manual face sorting), while other parts will be executed mainly on the graphics-card (i.e. rendering of triangles and face culling). We also find that in some cases it might be faster to render more faces and using a displaylist optimisation than rendering fewer faces without a displaylist.

### 3.5.2.1 Perspective Correction

As already mentioned in the previous Section, we sort faces into front and back-facing to minimise geometric redundancy (and consequently lighting calculations). One problem that exists with this method and that has already been hinted at in Section 2.2.3 is that of artefacts being introduced due to the approximation made for determining the orientation of triangles.



**Figure 15 - Face sorting without perspective correction  
at different distances to the object**

While this happens only rarely, it can produce undesired holes in an object (see Figure 15 for an example) and should therefore be addressed. To recap briefly, we consider the same view-vector for all triangles comprising an object in order to save the computational cost of casting a view-vector from the eye to each triangle individually. Technically, this situation holds only for a viewer infinitely far away from an object (Figure 16b). The approximation is still fairly good for objects that are relatively far from the viewer but fails when object and viewer are in close proximity (Figure 16a). In the latter case we assume the true

silhouette to be further back than it is perceived by the viewer under perspective projection. While this example shows well how the approximation can break down, it is not the silhouette that we are concerned about here primarily (the holes in the object are not due to the silhouette). Nonetheless, we will come back to discuss this issue a little further, later.

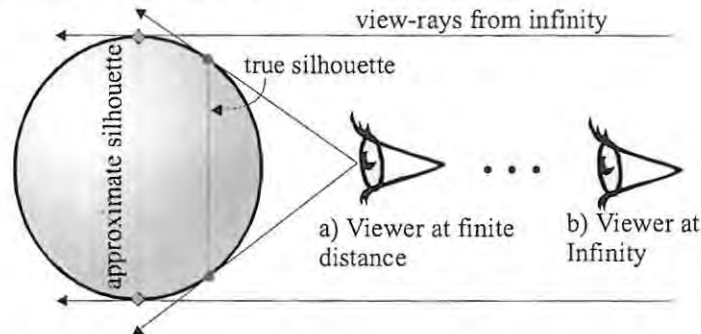


Figure 16 - View rays at (a) finite and (b) infinite distance from the object

To fully appreciate the situation and the origin of the holes in the object as shown in Figure 15, we illustrate the scenario in Figure 17. Firstly, we would like to point out the difference between the view-vector and *the direction that the viewer is looking at*. The latter certainly has an influence on how the scene is displayed, but it is important to notice that it does not influence which surface elements point towards or away from the viewer. This is solely determined by the viewer's position relative to the surface element. This is why we use the approximation to define the view-vector as

$$\vec{V} = C - V$$

where  $C$  is the centre of the Bounding box of the object and  $V$  is the position of the viewer. If the viewer is close to the object and near the edge of the bounding box, we can get the following situation: A surface element  $S$  close to the edge of the Bounding box has a normal  $N_s$  that when compared with the normal of the view-vector  $N_v$  appears to point backwards, where in reality it does point forward (a fact that can easily be verified by drawing a vector from the viewer to the surface element).

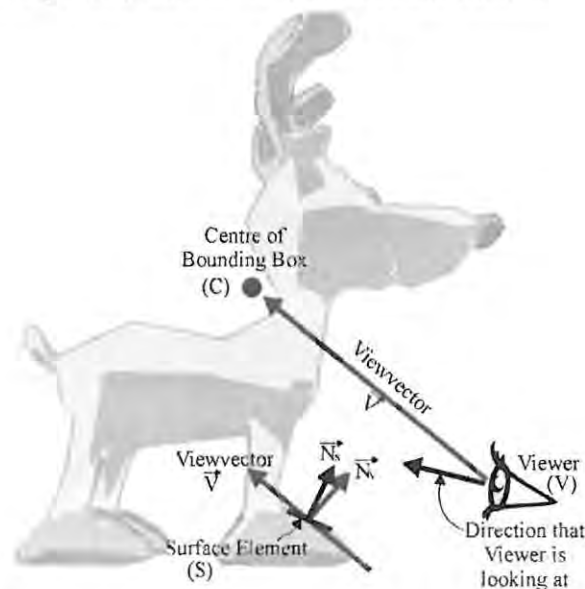


Figure 17 - Explanation for Object holes

In order to explain our solution to the problem, we use another schematic (Figure 18), which better shows the angles involved. Here we place the viewer on the bottom edge of the Bounding box (the most extreme position for the viewer) where she would be able to move towards or away from the object, thus changing angle  $\alpha$  in the process. If we now consider a surface element at the very bottom of the bounding box and almost parallel to the lower edge (if it were indeed parallel, the viewer could not see it at all), we see that the surface element and the view-vector form an angle  $\beta$  which is less than or equal to  $\alpha$ . Thus, if we include all faces that deviate by at most  $\alpha$  from the standard front-facing criterion (Dot-product of view-vector with surface normal is negative) in the front-faces, we are able to fill any holes that might appear due to our approximation. As can easily be verified from Figure 18,

$$\sin \alpha = \frac{a}{|\vec{V}|}$$

where  $a$  is the side length of the Bounding Box (we take the largest side-length as the most conservative measure, which is the reason why we drew the Bounding Box as square) and  $V$ , as above, the view-vector. We call  $\alpha$  the *perspective angle* or *perspective correction angle*, because we offset our calculations by its value in order to counteract the effects of perspective projection.

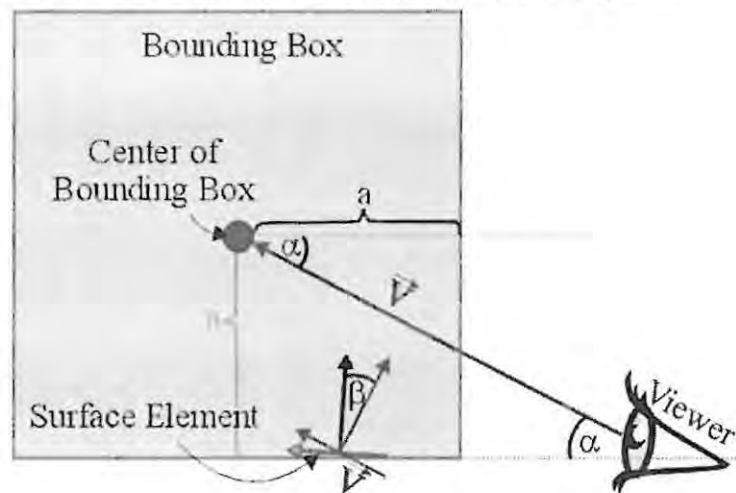


Figure 18 - Angles in an extreme case

The meaning of *offset* is shown in Figure 19. Figure 19a) depicts the standard case, where the dot-product of the view-vector and the surface-normal is shown as a cosine curve and negative values indicate front facing surface elements. At  $0.5 \cdot \pi$  the two vectors are parallel (N.B. the x-units of  $\pi$ ).

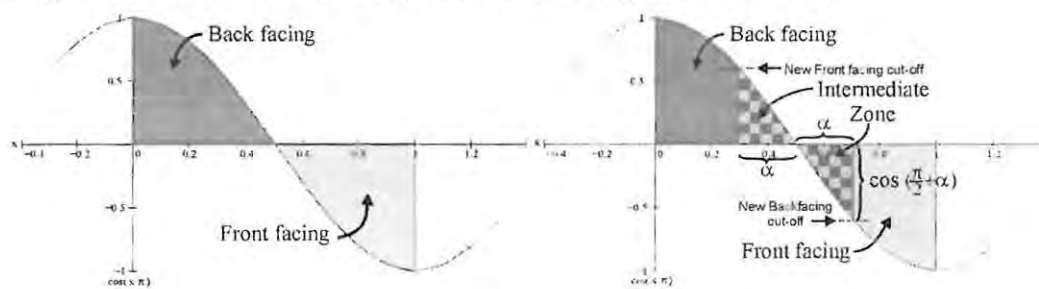


Figure 19 - Determining Face orientation: a) Standard; b) with correction offset



To accommodate for the perspective correction, we find the perspective angle  $\alpha$  as described above and take  $\cos(\pi/2 - \alpha)$  (which is equal to  $-\sin(\alpha)$  thus saving an addition) as the new front-facing cut-off value (instead of zero). This is illustrated in Figure 19b). We see that this results in more front-facing triangles being detected, which is what we needed to achieve.

Now, as promised, we return to the issue of the silhouette problem, which is due to the same approximation (and can therefore be remedied by the same approach), but which manifests itself in a different manner. As we showed in Figure 16, the true silhouette will lie much closer to the viewer if she is situated close to the object. The visual artefacts that appear in this case are shown in Figure 20a).

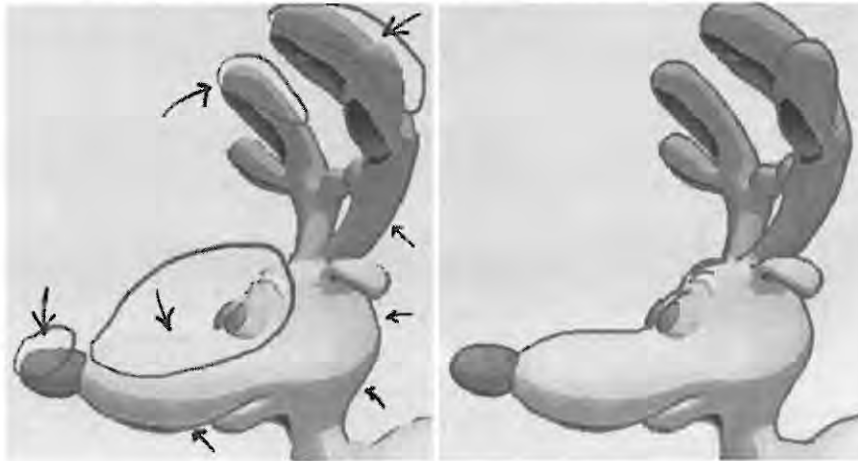


Figure 20 - Silhouette: a) Without perspective Correction; b) With perspective Correction

As we mentioned above, we could solve this problem with the same method and apply an offset of  $\alpha$  into the other direction in order to add more faces of the border region to the back-facing surface elements. Instead, we use a displaylist to render all back-facing elements, which is more than we actually need, but due to the hardware support for the displaylist still performs favourably. In the case where displaylists are not available, the perspective correction for back-facing surface elements will produce correct results.

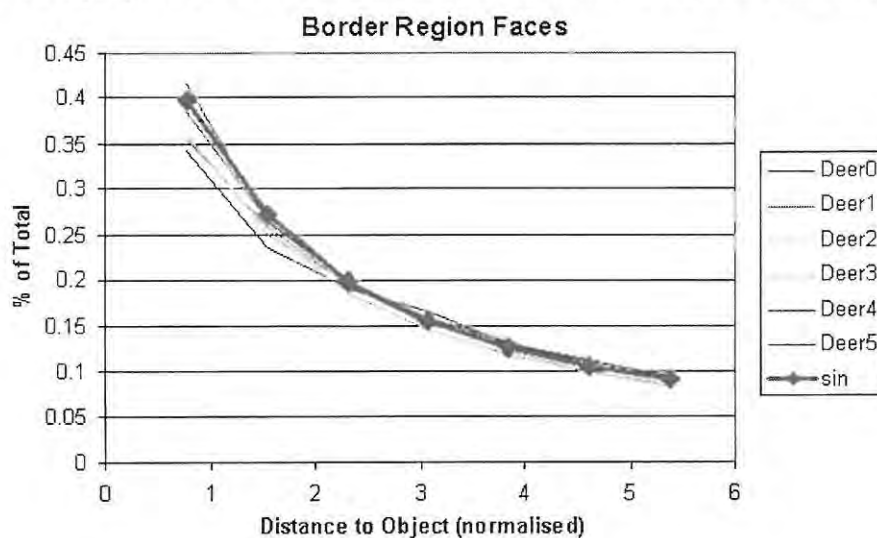


Figure 21 - Border Region Faces vs. normalised Distance

The overall result is as expected: As we move from far away towards an object, the number of front faces and the number of faces in the intermediate zone increase steadily. Figure 21 shows how the percentage of Border Region faces closely follows the probabilistic curve determined by the perspective correction angle (i.e. the sine curve). This means that as we get close to the object the total number of faces rendered increases and rendering performance drops (of course largely due to the fact that the object's screen dimensions become bigger and more pixels have to be filled). Altogether our approach successfully remedies the artefacts, which occur in some situations by dynamically adjusting the threshold, which decides if triangles are front- or back-facing according to the distance between the viewer and an object. While our method produces minimal computational overhead, the rendering performance may slightly decrease as a response to an increase in triangles to be rendered. The effect of this must be weighed against performing a perspective correct face-orientation determination or the loss of visual quality if artefacts appear.

### 3.5.2.2 Qualitative Results for Face-Orientation Determination

The face-orientation determination methods discussed in Section 3.5.2, apart from varying in efficiency, also produce renderings of different quality. Table 6 and Table 7 detail our findings with respect to the effects of the different orientation methods on the visual quality. The first row in these tables shows which geometric data is used to determine face-orientation ( $N_T$ .View means the normals of the triangles are used;  $N_V$ .View means the normals of vertices are used). The next row is labelled ODT (Orientation-Determination Traversal) and indicated in which traversal the actual orientation of the face is determined. All of our approaches use multi-traversal techniques to minimise computational overhead so that computation of data required for the orientation decision and the actual determination may be located in different traversal loops. The third row shows what condition is used to determine if a surface element is front-facing or not. In the fourth row we specify the visual quality, which depends on the number of visible holes that are produced by a given approach. The last row explains which artefacts (if any) are apparent for a given method.

Method	$N_T$ .View	$N_T$ .View	$N_V$ .View	$N_V$ .View
ODT	Triangles	Triangles	Vertices	Vertices
Condition	No persp. correction	With pers. correction	Mark all infaces as front-facing	Mark some infaces as front-facing
Quality	OK	Perfect	Perfect	Bad
Comment	Typical persp. correction artefacts	No artefacts	No artefacts	Many holes and/or flickering holes

Table 6 - Visual Quality vs. Face Orientation Methods (1)

The first column in Table 6 shows the standard approach, which calculates face-orientation per triangle, but without perspective correction. The visual quality was specified as OK, because artefacts are few and only visible for special views, close to the object and on special objects. Nonetheless, the artefacts do not depend on object-resolution (apart from the fact that the screen-area affected will be smaller for more detailed objects), i.e. will always be visible for the same object at different resolutions. We find this unacceptable and therefore always use perspective correction for all our other approaches. As expected,

the visual quality with the same approach but also using perspective correction is perfect (i.e. no holes in the object under any circumstances). The remaining approaches in this table and the next are based on the *Nv.View* approximation (i.e. determining the orientation of a face by the orientation of its vertices). The third column in Table 6 lists the results for marking all attached triangles of a given vertex as front-facing, if the vertex is front-facing (i.e. during the vertex traversal). As this is a very conservative approach, the visual quality is again perfect. The statistical method used to reduce the number of triangles marked per vertex on the other hand produces bad results with many holes for the static version and fewer holes (but flickering) for the random version.

Method	<i>Nv.View</i>	<i>Nv.View</i>	<i>Nv.View</i>
ODT	Triangles	Triangles	Triangles
Condition	All front-facing	At least two ff	At least one ff
Quality	OK	Almost perfect	Perfect
Comment	Some holes in low-detail objects	One hole in lowest detail object from special viewpoint	No artefacts

**Table 7 - Visual Quality vs. Face Orientation Methods (2)**

Table 7 lists the visual results for calculating vertex-orientation in the vertex traversal loop, but determining face-orientation during triangle traversal. The difference between the approaches listed is the condition used to determine front or back-faces. The first column is the least conservative approach and recognises a triangle as front-facing if all three of its vertices are front-facing. The visual quality is judged OK, because artefacts appear only for low-resolution objects, for which the vertex-approximation is far less accurate than for high-resolution objects. For high-resolution objects, the visual quality becomes perfect. The second column's condition is that at least two vertices of the triangle have to be front-facing. This condition holds true for more triangles than in the previous column and produces an almost perfect result: We could only find one hole in the lowest detail object from a special view-point. The most conservative condition that only one of the triangle's vertices needs to be front-facing produces perfect results for all objects under any circumstances.

### 3.5.2.3 Quantitative Results for Face-Orientation Determination

In Section 3.5.2.2 we have looked at the qualitative effects of different Face-orientation determination methods. In this Section we have a quantitative look at the number of triangles detected by any given method and how each method performs in terms of frames per second.

Method	$N_T$ .View		$N_T$ .View		$N_V$ .View		$N_V$ .View	
ODT	Triangles		Triangles		Vertices		Vertices	
Condition	No persp. correction		With persp. correction		Mark all infaces as front-facing		Mark some infaces as front-facing	
Quality	OK		Perfect		Perfect		Bad	
	FF	FPS	FF	FPS	FF	FPS	FF	FPS
Deer0	360	334.39	482	284.63	558	382.90	438	377.04
Deer1	522	248.66	717	213.00	841	288.35	668	311.97
Deer2	752	186.80	1021	150.40	1219	212.54	964	232.97
Deer3	1218	123.95	1687	95.06	1997	140.27	1608	153.52
Deer4	2992	54.16	4227	41.91	4873	61.27	4024	67.91
Deer5	5399	30.48	7802	22.39	9222	34.31	7795	37.61

Table 8 - Performance vs. Face Orientation Method (1)

Table 8 and Table 9 are again structured as in Section 3.5.2.2 and we included the Quality-verdict as a reminder, but this time we extend the tables with two columns for each method listing the Front-faces (FF) and frames per second (FPS) of each test-object.

Several facts are interesting about the number of Triangles determined as Front-facing. Comparing methods 1 and 2, we find that using our perspective correction approximation, we include between 33-44% additional triangles, which is a large amount considering the relative size of artefacts on visual inspection. We explain this large increase by having made the most conservative estimate about the perspective correction. To show the flipside of this issue, we included a new column in Table 9 (first column), which lists the values for performing correct view-vector calculations on each triangle. The number of Front-faces listed here are therefore the true number of front-faces under perspective correction not using any approximations. While we obtain the least number of front-faces, we also have to deal with the lowest frame-rate of all approaches. Even though less front-facing triangles have to be rendered, the cost of establishing the orientation of faces is so high that it is faster to render additional triangles. The idea is therefore not to strive for the lowest possible triangle-count, but instead to minimise the number of triangles as much as possible with as little effort as possible. This is the reason why methods 3 and 4 in Table 8 perform better than the previous ones, because calculations are performed per vertex and not per triangle. We end up with a lot more triangles to be rendered, but we can determine them a lot faster. Another interesting fact is that even though methods 2 and 4 are very close in face-count, their visual results are extremely different.

We quickly found out that in methods 3 and 4, a lot of time is spent on marking the same triangles as front-facing, so we proceeded to mark only vertices and then decide at triangle traversal time whether a given triangle is front-facing or not.

Method	N <sub>T</sub> .View		N <sub>v</sub> .View		N <sub>v</sub> .View		N <sub>v</sub> .View	
ODT	Triangles		Triangles		Triangles		Triangles	
Condition	View-vector/Triangle		All front-facing		At least two ff		At least one ff	
Quality	Perfect		OK		Almost perfect		Perfect	
	FF	FPS	FF	FPS	FF	FPS	FF	FPS
Deer0	299	212.19	387	420.26	503	417.70	558	412.10
Deer1	461	162.57	631	345.85	771	331.56	841	324.17
Deer2	632	119.40	903	265.24	1094	254.75	1219	248.29
Deer3	1050	71.33	1559	181.70	1816	164.38	1997	168.92
Deer4	2555	31.72	4133	81.32	4543	76.28	4873	76.74
Deer5	4843	16.70	8019	44.00	8680	42.23	9222	41.85

Table 9 - Performance vs. Face Orientation Method (2)

The results of this approach with the different orientation criteria are shown in methods 2-4 in Table 9. We can see that all rendering times are fairly close but the face-count varies between 15-44%. As we want to guarantee a perfect visual result and the frame-rates are so similar, we have no trouble opting for the most conservative case (method 4). In conclusion, we were able, by means of restructuring our algorithm and using several approximations, to increase rendering speed by 94-151% (a greater speedup gained for more complex objects), even though we increased the number of Front-facing triangles to be rendered by 86-90%. This means that the cost of rendering a triangle is so low that the per-triangle computational cost in our algorithm is considerable.

### 3.5.3 Anti Aliasing

Another (more visual) optimisation can be achieved by using anti-aliased lines with `glEnable(GL_LINE_SMOOTH)`. Unfortunately this operation is not always implemented and where it is, performance usually suffers greatly as this option is not widely used and therefore not usually optimised. An alternative is the FSAA (Full Screen Anti Aliasing) which some graphics card support and which smoothes the appearance of the whole scene and not just the lines. While a performance hit can certainly be expected while enabling this option, it seems that most graphics systems optimise their FSAA more than the line anti-aliasing. The level at which the FSAA actually happens also appears implementation dependent. In one of our experiments for example, anti-aliasing was performed on the screen output, but was lacking from memory-shots taken from the colour buffer.

### 3.5.4 Relative Rotation

Instead of multiplying each normal with the current model-view rotation to obtain the current orientation of the normals (as was done in the original code by Landers for ease of readability), we orientate the light vector by the inverse rotation, in effect achieving the same result, without having to perform matrix multiplication on each normal



### 3.6 Extensions

In the following Sections, we investigate various methods to extend the standard comic renderer to make it visually more interesting. More precisely, the existing lighting model is extended, the use of multi-coloured objects and silhouettes is elaborated upon and we discuss the use of textured objects.

#### 3.6.1 Extending the Lighting Model

$$I = k_d I_a + \frac{I_p}{d + d_0} [k_d (N \cdot L) + k_s (R \cdot V)^n] \quad , \quad \vec{R} \cdot \vec{V} = (\vec{L} - 2(\vec{N} \cdot \vec{L})\vec{N}) \cdot \vec{V}$$

Equation 1 – a) Phong reflection model; b) computation of dot product (R,V)

Next, we explore the effect of including a specular component in our lighting calculations. According to the Phong reflection model (Equation 1a) the specular component of light interaction is proportional to the dot product of R and V (which in turn depends on N, L and V – see Figure 5a) for an explanation of these Vectors) to the power of some number n, called the *shininess* value. As becomes evident from Equation 1b) the computation of this value involves a vector addition, two dot products and two further floating point multiplications (not including the additional overhead for the power of n). Altogether a fairly large overhead considering that the equation needs to be evaluated for all front-facing vertices (see Section 3.5.2 for a definition of this term). Fortunately, our optimised face-orientation algorithm helps us in this case. By multiplying the right-hand side of Equation 1b) out, we get

$$\vec{R} \cdot \vec{V} = \vec{L} \cdot \vec{V} - 2(\vec{N} \cdot \vec{L})(\vec{N} \cdot \vec{V})$$

Equation 2 - Expanded R dot V product

The first term on the right-hand-side of Equation 2 is not vertex-dependent and can therefore be computed once-off for a given rendering pass. The second vertex dot-product is our familiar N.L dot-product, which we have already computed. The last dot-product is new, but exactly what we use to determine the orientation of a vertex and therefore already computed. This means that what first appeared as a major overhead in terms of extra computation, turns out to come for *free* with our new face-orientation technique. It should be noted that for large values of the shininess value n approximations or optimisation should be used to limit the overhead of the power computation.

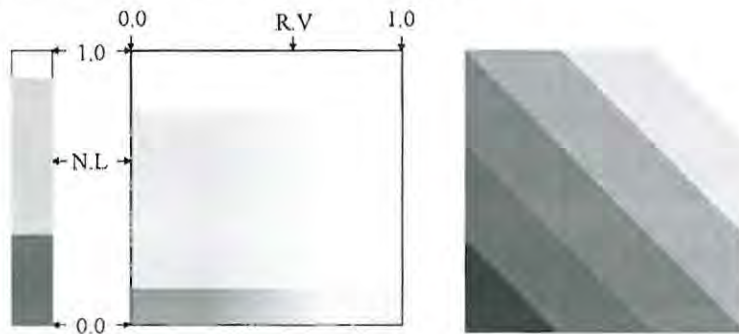
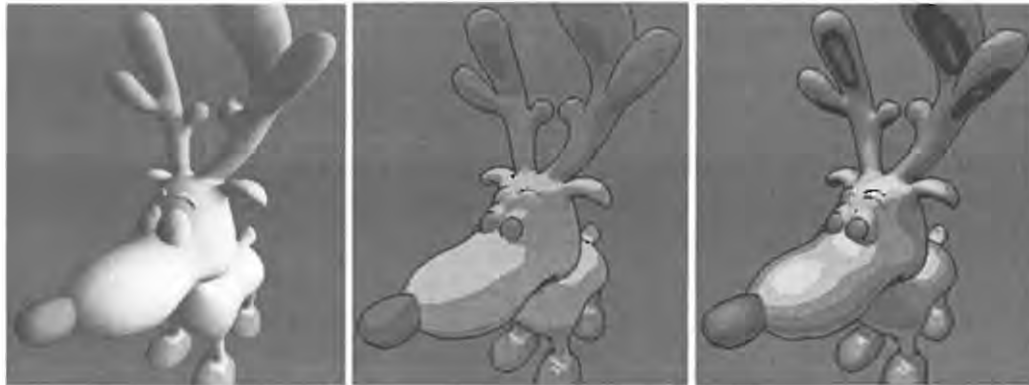


Figure 22 - a) 1D shade texture; b) 2D shade texture; c) example map

The way we make use of our extended comic shader is by using a 2D shading texture instead of a one-dimensional one. Examples of such a texture are depicted in Figure 22b) and c). Figure 23 shows how our extension effects the rendering result. Figure 23a) demonstrates how the Deer object looks with standard OpenGL shading applied. Figure 23b) is shaded using a 1D texture (the texture was obtained from Figure 22c) by setting the R.V component to zero) and Figure 23c) shows how our lighting extension to the standard comic renderer introduces specular highlights, while still maintaining its comic-style appearance. Animation B demonstrates the effect added by the specular component by showing the same object rendered in both the standard and extended comic style.



**Figure 23 - a) Standard OpenGL shading; b) standard comic style; c) extended comic style**

While we believe that the visual output of our extended style is more interesting and interactive (this is due to the fact that the viewer's position enters into the lighting equation – as opposed to the standard approach, where the viewer's location is irrelevant), we can also achieve other visual effects with our method by simply modifying the shade texture. Animation C illustrates both these facts, by demonstrating view-dependence using a metallic shade-map. A variety of different effects can be produced in this fashion by altering the shade texture without the need for alterations to the main (extended) algorithm, which is listed in Listing 4.

```

Let L = Light Vector
Let Nx = Normal (at any of Vertex, Triangle, etc.)
Let R(x,y) = Reflection Vector of x on surface with normal y
Let View = View Vector

Render
// calculate customised lighting
// requires 2 dot products and one vector subtraction per iteration
Compute (L.View)           // used in lighting calculations
Compute angleOffset        // used for perspective correction
For each Vertex V do
  If (Nv.View < angleOffset) // if vertex is front-facing
    calculate( Nv.L )
    calculate( R( L,Nv ).View )
    setFrontFlag(V)
  Else
    setBackFlag(V)
  End If
End For
Next Vertex

For each Triangle T do // render solid interior
  If (Condition (V1(T),V2(T),V3(T))) // if front-facing rule applies
    Draw T // Draw triangle in fill-mode with shading applied
           // through the use of a texture-map that is indexed
           // by previous lighting calculations.
  End If
End For
Next Triangle

For each Triangle T do // render silhouette

```

```

        Draw T           // Draw in line-mode with thick dark lines. Since no
Next Triangle          // viewer or light-dependence, use displaylist to
                        // optimize performance
End Render

```

#### Listing 4 - Extended Comic style Algorithm

There are several reasons why we draw each triangle for the silhouette and not just the back-facing ones. We could easily extend the “if-statement” in the first triangle-traversal and add an else clause to render a back-face otherwise. But as we discussed above, a perspective correction has to be applied to the back-faces as well, so a simple “else” would not suffice. Yet another condition would solve the problem, but more importantly, a variety of state-changes would be necessary in order to switch the OpenGL engine from our comic specific fill-mode to the silhouette specific line-mode. OpenGL state-changes are very expensive operations and should therefore be kept to a minimum. The next option would be to traverse the triangles again, but perform a test to render only the ones that are back-facing. It turns out, that using a displaylist, which renders all triangles, is approximately as fast as determining back-facing triangles manually on our test set-up. In general, this fact has to be established for any given system and the fastest method can be chosen.

##### 3.6.1.1 Qualitative Results for Lighting model extension

By extending the lighting model of the standard comic renderer, we increased its computational complexity and we had to use various approximations (as well as remedies for these approximations) in order to make the extended comic renderer perform as well as the standard comic renderer. To justify all these efforts, we now demonstrate the visual effect produced by the extension and compare it to a physically more correct version.

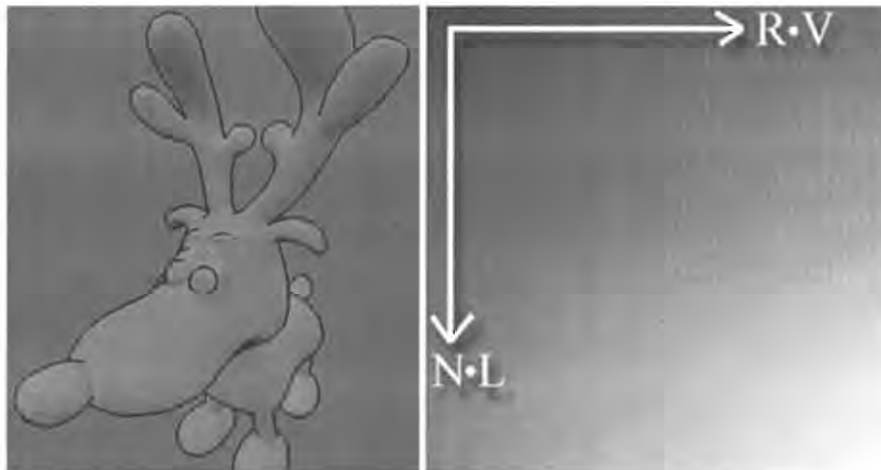
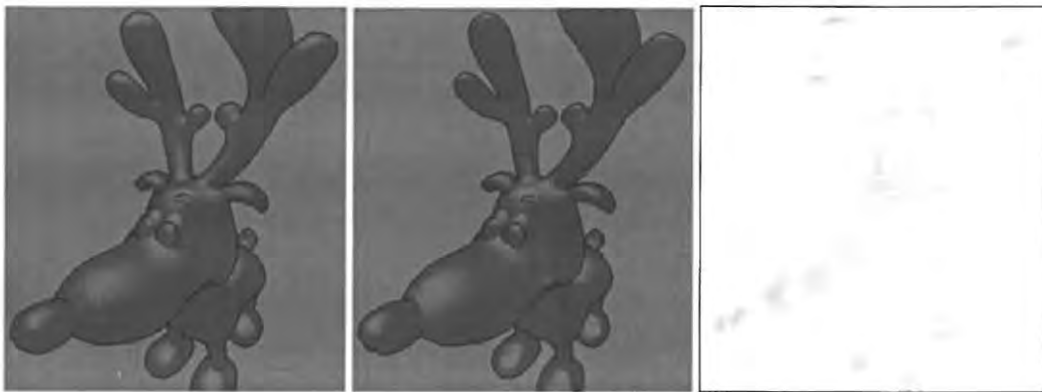


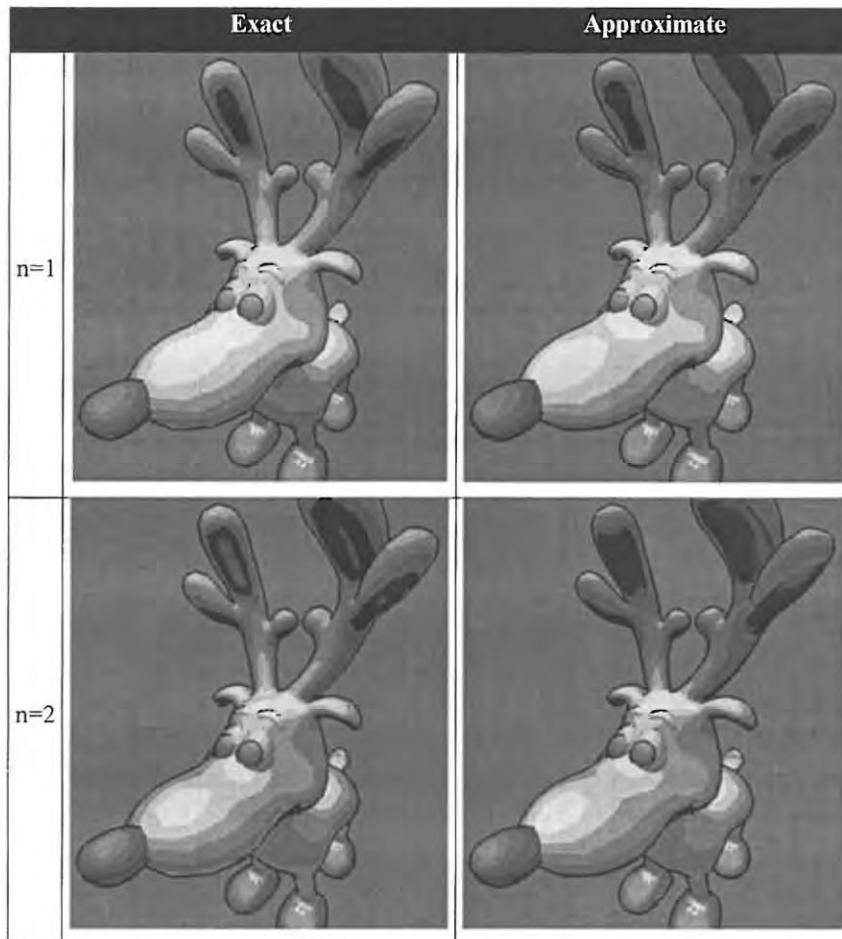
Figure 24 - Extended Light Model: a) Diffuse Component; b) Shade-Map

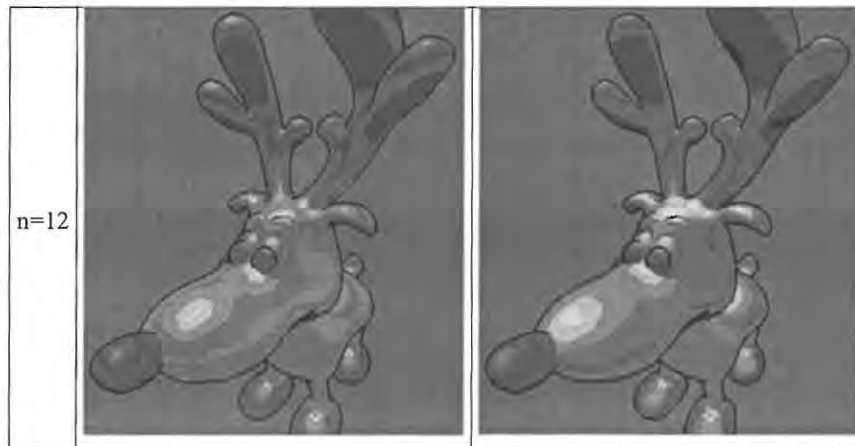
In Figure 24b) we show a smooth shade-map with a red specular component and green diffuse component. These are combined additive over the map to produce white at the point where specular and diffuse components are at a maximum and black where both are zero. Figure 24a) therefore shows the diffuse component only (specular component is zero). This diffuse component is only dependent on the curvature of the object and the light direction, so that view-vector approximations will have no influence on this component.



**Figure 25 - Specular Component: a) Exact; b) Approximate; c) Difference**

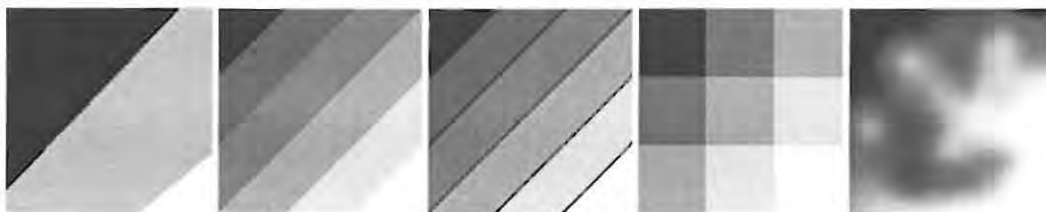
In Figure 25 we show the specular component only, with the diffuse term set to zero. The shininess value is set to 6, producing fairly sharp specular effects. Figure 25a) shows the more accurate version, which uses an exact view-vector calculation for each vertex. Figure 25b) on the other hand uses our constant view-vector approximation. As becomes evident from Figure 25c), where a difference-image of the two versions is shown, the two results are not identical. Nonetheless, the visual difference is small enough that the untrained eye would have difficulties deciding as to which version is more accurate. As expected, differences occur for any given shininess value, but the visual results are always similar.





**Table 10 - Comparison of results for various Shininess values**

Table 10 shows the visual results for the correct view-vector calculation and our approximation side-by-side for different values of the Shininess value  $n$ . This value determines how *glossy* an object appears. Our approximation works best on high curvature regions and fails completely on a perfect plane. This explains why some surface areas of the approximate version resemble the correct version more closely than others. From our experiments we find that our approximation suffices for any situation, where the effect of specular reflection is important (as opposed to a physically more correct approximation) and for objects that are non-planar.



**Figure 26 - A Collection of Shade-maps (a-e)**

Figure 26 shows various shade-maps that we have experimented with. Even though all of these are greyscale, we have shown above (Figure 24) how a colour-shade-map can be used to simulate coloured light-sources. Introducing black stripes into the shade-maps to separate the colours results in different shade-regions being separated by black lines as shown in Figure 28a), which uses shade-map c. As mentioned above, a large variety of effects can be produced simply by altering the shade-map. Figure 27a) for example shows a monochrome, very flat rendering of the statue of liberty using map a, whereas map e produces the metallic-silver finish of Figure 27b). In the same Figure we can also examine how the specular approximation works well for everything except the book that Liberty is holding, due to its planar nature.





Figure 27 - Statue of Liberty: a) Monochrome; b) Highly reflective Metal shading

The Blender in Figure 28b) uses map d, creating an interesting effect of shades crossing over, due to the checkerboard layout of map d. In essence, we can re-produce anything from a close approximation of the common Phong-shading model to a totally flat-shaded object through our shade-mapping technique.

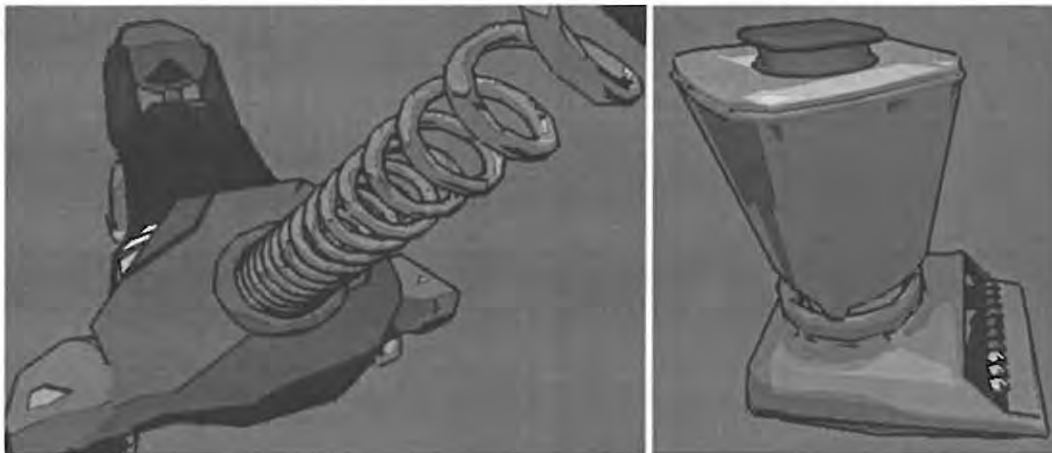


Figure 28 - More examples: a) Tail of Slinky the Dog; b) A Blender

### 3.6.2 Using existing Colour Information

While the original algorithm by Landers uses one uniform colour for the whole object, we think that the visual result is much more interesting if we make use of existing colour-information (see Figure 14b)). By setting the texture environment to modulate relevant pixels, we are able to use existing vertex colours (or base-textures) and shade them accordingly. We make sure that each triangle is coloured with exactly one colour (even if the three vertices of that triangle had different colours associated) to ensure that no colour bleeding occurs across the triangle (this will ensure a flatter, more comic-stylish look).

### 3.6.3 Silhouette Colour

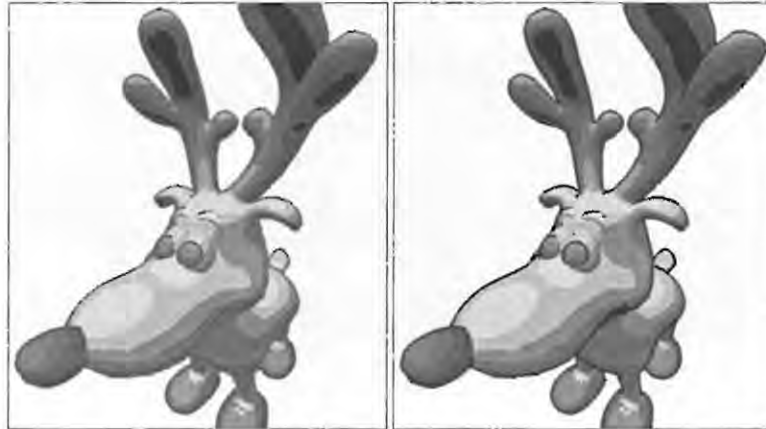


Figure 29 - Silhouette Colour: a) Varied; b) Single

Other simple variations are easily implemented. Comparing Figure 29a&b) for example one can observe a difference in the silhouette style. The deer in Figure b) uses a plain black outline for the silhouette and other edges, while the deer in Figure a) actually use a darker version of the object colour. Rendering the edges with the colour of the defining vertices and scaling the colour down by a certain factor very easily achieves this.

#### 3.6.3.1 Quantitative Results for using Silhouette Colour

The effects of the remaining visual enhancements are shown in Table 11, where we compare frame-rates of the Extended Comic renderer with those of the Extended Comic renderer with Silhouette colour and Base-texture (multi-texturing) respectively. It turns out that using multiple colours for the silhouette displaylist incurs a negligible penalty especially considering that the displaylist could still be optimised to accommodate multiple colours. Using multi-texturing on the other hand results in a more severe performance penalty as the frame-rate drops between 30-32%. Still, real-time performance is very comfortably achieved even in this case.

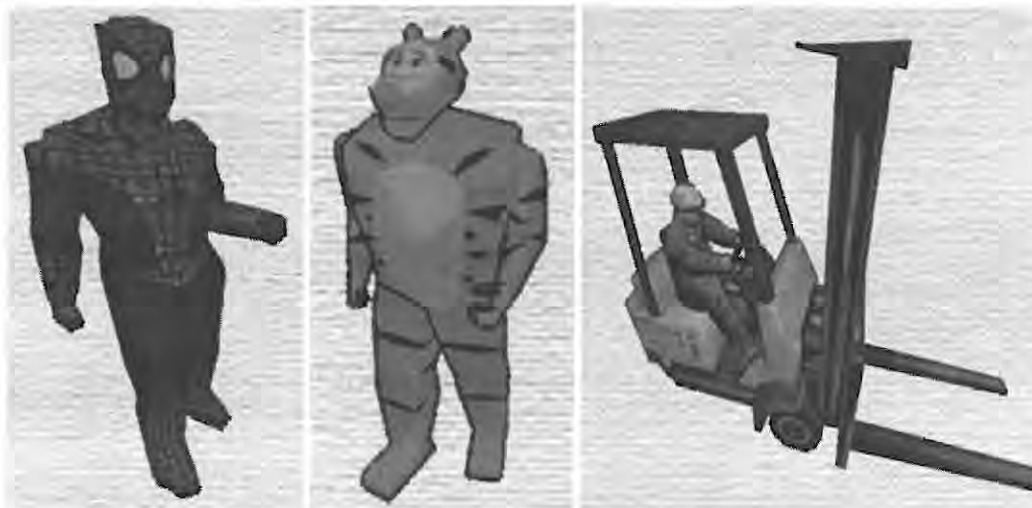
Name	Extended Comic	Coloured Silhouette	With Base Texture
Deer0	412.10	406.33	314.12
Deer1	324.17	317.04	264.01
Deer2	248.29	242.04	199.72
Deer3	168.92	164.82	135.21
Deer4	76.74	75.09	57.69
Deer5	41.85	41.07	31.04

Table 11 - Performance with Silhouette Colour and Multi-texturing in FPS

#### 3.6.4 Dealing with existing Textures (Multi-texturing)

Objects with simple colour schemes like the dog in Figure 14b) or the reindeers in Figure 23a&b) produce a convincing and effective comic-style look. A large variety of objects which can be downloaded from the

internet or are available in commercial packets use textures instead of colour information (even though the two are not mutually exclusive) to add detail to their geometric models. In fact the common trend amongst game designers is to employ objects of minimal polygon count and furnish them with textures of considerable detail – in most cases an adequate compromise. Naturally, our extended comic renderer needs to be able to cope with the large number of objects in this class. The basic problem that needed solving was that we already were using a shade-texture, so that an additional texture could not be applied. The easy and straightforward solution we found is to use multi-texturing. OpenGL 1.2.1 (see [83], p.240) defines the `GL_ARB_multitexture` commands (which up to that point were optional extensions) and at least four texture units are supported by the GL engine (this does not mean that a specific vendor has to implement this many – most multi-texture capable cards today only support two simultaneous texture units). By enabling blending, assigning one texture unit to the rendering of the object texture and another one to rendering the shade texture, we can in fact produce our comic style for textured objects. Of course the same results could be achieved on systems without multi-texturing by performing a separate rendering pass for each of the textures to be used. For this to work the depth-buffer has to be set to `GL_LEQUAL` in order to come through on the second pass. The incurred performance loss is then proportional to the number of polygons that have to be rendered twice.



**Figure 30 - Comic style with multi-texturing: a) Spiderman; b) Tigger; c) Forklift**

As we show in Figure 30, the quality of the output largely depends on the object's base texture. Figure 30a&b) have fairly flat textures to begin with so that the shading applied by the comic renderer accentuates this fact in a complementing fashion. Figure 30c) on the other hand exhibits so much object texture detail, that the subtle effect of our comic-shading gets lost (this is true for still images anyway, animations are able to show how shadows and lights sweep over an object, so that the static object texture can be distinguished from the dynamic shade texture). We therefore conclude that for most effective results in connection with rendering textured objects in comic-style these textures should be as flat and comic-like as possible to begin with. One way of achieving this at run-time would be to decrease the number of colours used in a texture via an image filter. The limitations of the image filter then determine the visual outcome of this approach.

As already mentioned in Section 2.3.1, the number of state changes should be kept to a minimum. As we are faced with objects of various makes and origins, we cannot guarantee that all faces of our objects are textured (even if some of them are). This poses problems when it comes to rendering these faces as no texture or texture vertices may be defined for these objects. Several solutions exist to this problem:

- Sort faces into textured and non-textured bins and render them separately
- Define default (empty) texture and default texture vertices for non-textured faces
- Turn off base texture unit for each non-textured faces

The last method involves the most state changes and is therefore not advised. The second method involves creating a further empty texture (which can be as small as possible, i.e. 1x1 or 2x2, depending on the exact interpretation of the OpenGL texture limitations). Frequent texture rebinding may create an unnecessary overhead, which can be avoided if non-textured faces are rendered together. The first method implies a slight modification in the rendering algorithm, but bin sorting can be performed at load-time, as texture assignment of faces does not change at run-time. This solution requires the least state-changes and multi-texturing can even be disabled for parts of the faces, which depending on the implementation, may speed up rendering performance.

## 3.7 Results

### 3.7.1 Comparison of Approaches

In Table 12, we list the performances of the *Default renderer* (see Section 2.4.3), the *Standard Comic renderer* (Section 3.4) and our *Extended Comic Renderer* (Section 3.6.1) in absolute as well as relative terms.

Name	Faces	Absolute (fps)				Rel. to Default		Rel. to Def.+Text.	
		Default	Def. + Text.	Std. Comic	Ext. Comic	Std. Comic	Ext. Comic	Std. Comic	Ext. Comic
Deer0	578	530	481	454.04	412.1	0.86	0.78	0.94	0.86
Deer1	870	508	443	347.35	324.17	0.68	0.64	0.78	0.73
Deer2	1284	480	426	255.62	248.29	0.53	0.52	0.60	0.58
Deer3	2162	435	400	175.22	168.92	0.40	0.39	0.44	0.42
Deer4	5488	282	275	76.63	76.74	0.27	0.27	0.28	0.28
Deer5	10648	170	167	41.01	41.85	0.24	0.25	0.25	0.25

Table 12 - Performance Comparison of Default Renderer vs. Standard Comic and Extended Comic

The performances for the Default Renderer are given in two variations: One being the fastest possible version on our test-platform, while the other textures our test-objects. The reason this is mentioned is because the deer-test-objects do not originally have a texture, but since both of the Comic renderers use texturing it is only fair to compare their performance against the default renderer using texturing as well. As can be seen from column 3 and 4 in Table 12, the difference between applying a texture and not is fairly small (between 2-10%, with greater difference for less detailed objects), but should be noted. In

columns 5 and 6 we list the absolute performances of the Standard Comic approach and our Extended Comic approach respectively. It is evident that the two perform equally (always within 10% of each other), the Standard Comic renderer performing favourably for low-resolution objects, while the Extended Comic renderer has a very slight advantage for high-resolution objects. We attribute this to the fact that modern hardware accelerated graphics cards are so extremely fast. The Standard Comic renderer, which uses a brute force method to render triangles performs well if there are a limited number of triangles to be rendered. The Extended Comic renderer uses various calculations to limit the number of triangles to be rendered, the initial cost of which diminishes its performance. For many triangles though this technique pays off as the graphics pipeline gets saturated. This is the reason why we are able, with our various improvements to the Standard Comic algorithm, to extend the standard lighting model by a (theoretically) expensive, view-dependent component and still perform basically as well as the standard version. Similarly, we explain the results obtained in columns 7-10, where we compare the performances of the Comic renderers against those of the Default renderer. As the Default renderer makes use of 100% hardware acceleration, it is mainly limited by the speed and width of the graphics pipeline. The comic renderers on the other hand have to perform various calculations per vertex and/or triangle. This is the reason why their performance decreases with increasing number of faces. In absolute terms, we are very pleased to see that both Comic renderers fulfil their goal of real-time performance.

### **3.7.2 Face-sorting vs. Displaylist in Extended Comic Renderer**

We stated earlier, that on our test-platform it is more efficient to render all triangles in order to generate the silhouette instead of just back-facing ones. This is interesting considering that the orientation-information is already available from the first rendering pass (drawing the fronts). It means that it is roughly twice as expensive to check the orientation of a triangle and then submit it as to render it with the optimised displaylist on the graphics card. We note though, that the factor is not exactly two, as there are usually more than half of the faces back-facing (using our perspective correction) and the relative performance of the two approaches is not equal to one. In Table 13 we list the details of our tests. Columns 3 and 5 show absolute frame-rates of the whole rendering stage, while columns 4 and 6 detail the time (in milliseconds) to render only the silhouette. Columns 7 and 8 show the relative performances. The conclusion is that even though the display-list version is usually slightly faster the overall performance is basically the same, indicating that the silhouette rendering is a minor factor in the total rendering loop.



Name	Faces	Face Sorting		Display List		Comparing	
		(fps)	Silhouette only (msec)	(fps)	Silhouette only (msec)	Relative	Backs Relative
Deer0	578		1.42		0.76		1.87
Deer1	870		0.97		0.85		1.14
Deer2	1284		1.39		1.17		1.19
Deer3	2162		1.72		2.00		0.86
Deer4	5488		3.77		2.84		1.33
Deer5	10648		6.86		5.21		1.32

Table 13 - Comparison of Face-Sorting vs. Displaylist caching (Extended Comic renderer)

### 3.8 Summary

In this Chapter, we define the characteristics that we consider vital for recognisable Comic Art, namely a heavy and dark silhouette as well as flat or very banded shading.

A problem statement is formulated, stating that a silhouette has to be identified and rendered as well as employing a shading method for banded shading. As the basis for our investigations we use an algorithm by Lander [43]. We develop various optimisations and extensions to this standard algorithm. A generic perspective correction approximation helps us to quickly determine orientation of faces and vertices. The orientation determination itself is discussed and we provide a very successful optimisation that improves both calculations of the necessary lighting values and determination speed of the face-orientation in one interlinked approach. The diffuse-only lighting model of Lander is extended to include a specular component. Even though this component is view-dependent and involves surface reflection, we are able with our optimisations to reach between 90-100% of the speed of Lander's basic approach. Extrapolating our results we predict that for a large enough number of triangles (>10000 on our test-system) our approach will actually outperform the basic approach. Both Comic renderers perform within 25-86% of the Default Optimal renderer meaning that even at the low-end of performance frame-rates of above 40 can be achieved. The drastic drop-off in performance for high-resolution objects is due to the computational overhead in performing custom lighting calculations. In the advent of graphics cards with custom per-vertex shading capabilities this performance-loss will be severely reduced.

Apart from extending Lander's lighting model, we also experiment with other visual extensions such as multi-coloured silhouettes, comic shading on textured objects and vertex-colour information. The results of these investigations are that vertex-colour can be used to greatly increase the visual appeal of comic objects, a coloured silhouette provides an interesting and pleasant alternative to the common black outline and base-textures can easily be incorporated into our algorithm using multi-texturing. It is also our finding that the effect of comic shading is easily hidden when base-textures are visually too complex.

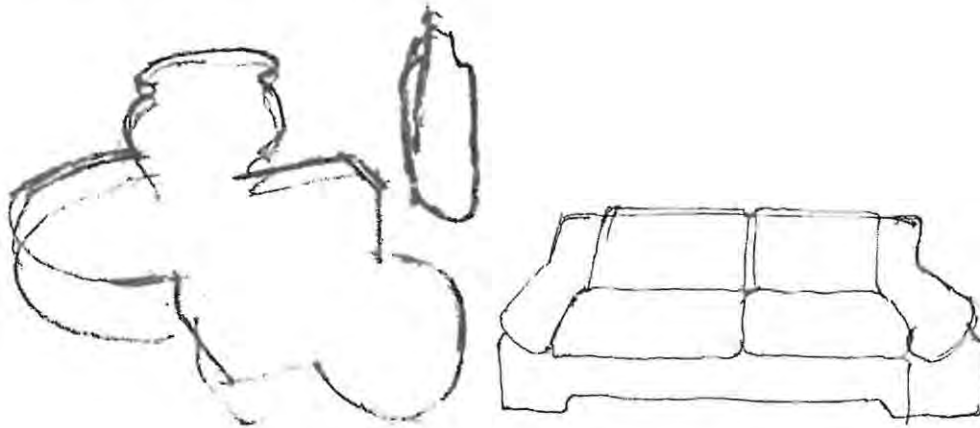
---

In conclusion we have successfully implemented a high-quality real-time comic renderer capable of rendering between 40 and 400 frames per second. In addition to that and through the inclusion of a specular light component into the algorithm, we have created a generic shading-mapping method that does not only cater for comic rendering, but is able to create a multitude of other effects as well, all the way up to the standard Phong shading, by mere choice of a suitable shade-map.

## 4 Sketching

### 4.1 Introduction

#### 4.1.1 Definition



**Figure 31 –Some sketch examples: a) Outline to be coloured in {Charcoal};  
b) Composition example {Ink} (both [87])**

Sketching is used in many real-life situations (like illustrations on a chalk board, doodling on a paper, sketching a map for directions, etc.) and in most cases represents a simplified, line-art version of reality. Depending on the artist (novice, beginner, expert, etc), the drawing medium (pencil, pen, chalk, ink, etc.), the paper medium (paper, foil, chalkboard, canvas, etc.) and the purpose (demonstration, illustration, explanation, etc.) the outcome of a sketch can vary widely indeed. Figure 31 and Figure 4 show both common and extreme examples of sketching, respectively. As already mentioned in Section 1.4.2, abstraction, apart from extracting relevancy, can be useful in the rendering process, because it very extensively limits the necessary detail to be reproduced on the screen. Figure 31b), for example, shows very few horizontal, vertical and diagonal lines, none of which are particularly straight or accurate, but the concept of a *sofa* is understood immediately.

As with the comic style (Chapter 3) we therefore have to identify the key features that make a sketch recognisable as such. It becomes evident that the above-mentioned variables of sketching are poor qualifiers. We thus note the following features that are common to most sketches:

- Drawn by hand (Randomness / Uncertainty-Factor)
- *Economy of line* (little, but important object detail)
- Few colours used (monochrome)

The fact that most typical sketches are drawn by hand exposes itself in that they are usually imperfect. Figure 32 illustrates this fact with a typical sketching example. Straight lines are not perfectly straight,

lines do not always converge in common points and several lines may be drawn to approximate some average line in between. This is no necessity; as for example technical sketches (even though produced by manual means) may appear quite elaborate using rulers and compasses to produce exact geometric curves and forms, but even those may fail to realistically converge on object boundaries.

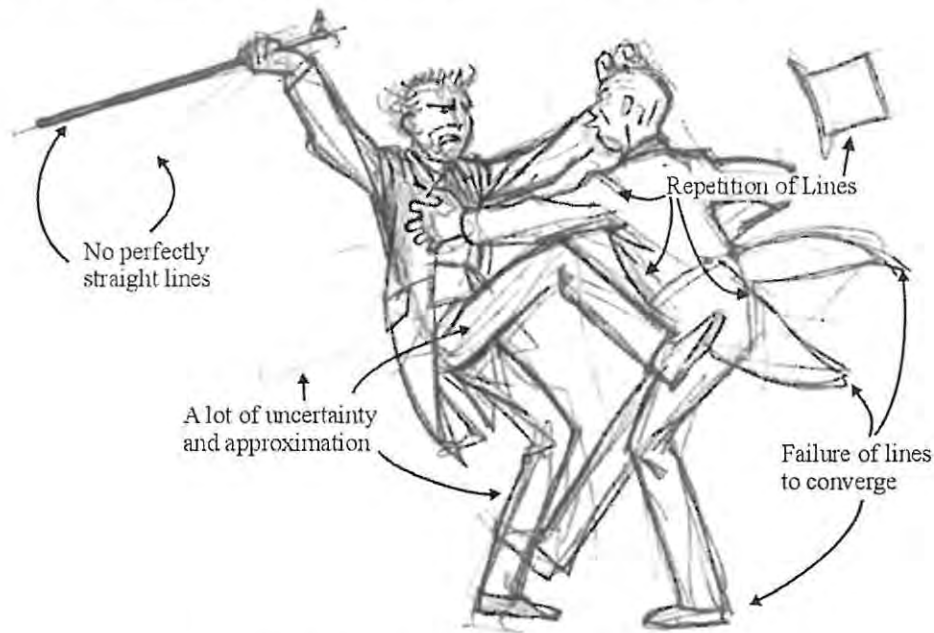


Figure 32 - Typical Sketch Example {Pencil} ([36])

Most sketches limit themselves to providing an outline of the real object with necessary detail where appropriate. This implies that a wealth of secondary information (like shading, colour, texture and other material properties including reflectivity and shininess) is reduced or discarded altogether in order to draw attention to object detail that is considered of primary importance. An example that is often quoted comes from Lansdown and Schofield [44] and deals with the situation a car mechanic finds himself in. He wants to repair an engine and therefore needs information about the structure and workings of that engine. An image of photorealistic quality would help him little, as he has the real engine in front of him. Another example would be directions to get to a certain location within a city. A photographic image from a satellite will most likely be of less help than a layman's sketch, where streets are simple lines and only the ones to be taken en route are drawn. It is for these reasons that most sketches suffice with an approximate outline of an object and only a hint of shading, where the three dimensional shape of an object is important.

## 4.2 Problems

### 4.2.1 Problem Statement

In accordance with Section 4.1.1, we must solve the following problems:

- Convey a *manual-production* look
- Identify important object detail (semi-) automatically
- Render the specific object-detail with:
  - Deliberate Imperfections
  - Rudimentary hints at shading if necessary

### 4.2.2 Implementation-specific Problems

In general, object-data is accurate and physically correct implying that imperfections necessary to imitate a manually crafted appearance are not part of the object-specifications. We therefore have to define and apply what we call *uncertainty functions* to the existing object-data in order to emulate some form of human error.

Due to the imperfection aspect of the sketching process and the animation aspect of the rendering process we have to address the issue of time coherence. While it would be unrealistic to expect an unaided manual sketch to render coherently between consecutive frames all the time, it will produce visually distracting images if all consecutive frames differ from previous ones (this is usually perceived as *flicker*). A compromise will therefore have to be made to find an optimal update-rate of the uncertainty data superimposed on object data. Alternatively, the uncertainty function has to be chosen to be temporally smooth. Hertzmann and Perlin [32], [33] state that an update-rate between 10-15 frames per second appears most natural to people. Whether this is due to the fact that the majority of people are used to these frame-rates from experience with stop-motion animation, or whether there are more fundamental reasons for this, is not mentioned.

As we stated above, the silhouette of an object is usually crucial to convey shape-information. Other folds and creases might be of importance to specify detail within the silhouette boundaries. Still, we do not want to limit ourselves to these geometric measures and we therefore invent the concept of an *importance-function*, which takes as input some object-data and produces as output how important this data is. In this light, the silhouette condition is just a specific importance-function, even though a dynamic and context-sensitive one (by being view-dependent).

If we want to use the silhouette condition as one of our importance-functions, we have to compute it from the object-data in an efficient manner. The reason why we cannot just simply use the silhouette generation algorithm of the Comic renderers (Chapter 3) is because there the silhouette is generated as a by-product of the OpenGL rendering process in image-space. In order to apply an uncertainty function, we need to



generate edge-information in object-space, before the OpenGL engine is engaged. In some cases other definitions of important object-data might be applicable (e.g. the different mechanical parts that make up a machine). Here, usually some form of world-knowledge must be available and applied in order to signify importance. Importance-function thus defined are generally static.

Rendering should be line-based in one form or other. Standard shading relies on colour-brightness variation, so we have to find an alternative way to convey shape-information through the use of appropriate line-placement, where necessary.

It is conceivable that any given solution to our problems might generate edges which should be hidden by the body of the object (these are called hidden lines). We must use some method of hidden line removal to address this issue. Again this can be seen as another context-sensitive importance-function.

### 4.3 Solution

In order to produce Line-Art drawings of objects it is convenient to work with the edges of an object. We therefore need to compute the importance function of the edges of an object for the current context. The meaning of context may be as simple as the current relative positions and orientations of object and viewer (i.e. view-dependence) but may also include information about parts of the object that should be hidden or highlighted. We then render a line-art representation of the object by drawing the important edges after randomisation (i.e. one or a set of uncertainty functions) has been applied to it. Shading information is best computed over faces (as opposed to edges) because the notion of a filled area is usually associated with it. We therefore need to draw the faces of the object in some kind of line-shading mode. We also may want to update the randomisation of the uncertainty functions every so often in order to create a dynamically changing, animated look. These steps are summarised in Listing 5.

```
Render
  // update the importance information in the given context
  Compute (importanceFunction (Object, Ei))

  // Render the Line-Art
  For each Edge E do
    If (important(E)) then
      Draw randomised(E)
    End If
  Next Edge

  // Render the shading
  If (shadingRequired) then
    For each Triangle T do
      Draw T // Render T with Line-Art style shading
    Next Triangle
  End If

  // update the randomisation function
  If (randomisationAnimationRequired) then
    Every nth frame do
      Update (randomSeed)
    End Every
  End If
End Render
```

Listing 5 - Generic Sketching Algorithm

#### 4.4 Standard Approach (Random Perturbation Sketching)

As a starting point we use the silhouette criterion as our importance function. This means that an edge is important if it is located on the silhouette (which in turn is context-sensitive, i.e. view-dependent).

Next we address the uncertainty function. Inspecting a variety of manual sketches (see Figure 32 for a prime example), we identify several observables. In Figure 33, we define the following uncertainty features (which are consequently parameters in the rendering process):

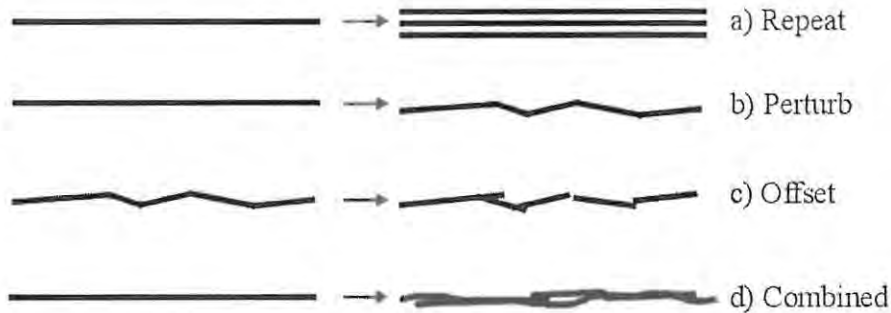


Figure 33 - Uncertainty functions

- One or several lines are drawn for each silhouette edge (a)
- Lines are subdivided into smaller fragments and randomly perturbed (b)
- Start and Endpoints of original lines are offset randomly from their original position (c)

The number of times each line is rendered has a well defined upper limit, which is lowered according to the total number of silhouette edges in an object and for a given view (as we use an exhaustive search method to identify edges, this information is readily available).

The random perturbation of silhouette lines was inspired by an algorithm to generate semi-random coast lines by Hill [34] and works in the following simple and recursive manner:

1. If a line  $L$  is longer than a given maximum length go to step two, otherwise render the line
2. Divide the line into two segments  $L1$  and  $L2$  by generating some mid-point  $M$  (need or should not be true geometric midpoint) so that  $L1 = \{\text{StartPoint}(L), M\}$  and  $L2 = \{M, \text{EndPoint}(L)\}$
3. Call step one recursively with  $L1$  and  $L2$

Hidden line removal is performed easily and effortlessly with the background-preserving method discussed in Section 2.3.2. A sample object rendered in this style can be seen in Animation D.

## 4.5 Optimisations

### 4.5.1 Level of Detail Optimisation

To limit the number of silhouette edges to be rendered, we implement two initialisation optimisations (i.e. constant, non-context-sensitive importance functions). As mentioned in Section 2.2.3, inface edges are of no visual value (Markosian et al [47] would argue that their probability of being part of the silhouette is zero) and they are therefore filtered out immediately. Similarly, we can filter out edges whose dihedral angle is above a certain threshold value, which we call *MaxAngle*. Furthermore, edges with lengths below a certain threshold value, called *MinLength*, are also disregarded.

#### 4.5.1.1 Quantitative Results for Level of Detail Optimisations

In Section 4.5.1, we discuss two simple level-of-detail (LOD) reduction methods:

- Discarding edges if their lengths are below a certain threshold value (*MinLength*)
- Discarding edges if their dihedral angle is above a certain threshold value (*MaxAngle*)

In this Section we have a brief look at how suitable these optimisations are for our purposes. In Figure 34 we show the edge length distributions of our six test objects. To obtain the graphs, we created 20 equal sized bins into which we placed edges according to length (the different maximal lengths for the different objects were taken into account). What we see for all of the graphs, is that there is an extreme sharp rise, which then tapers off slowly, meaning that most edges for any object are relatively short.

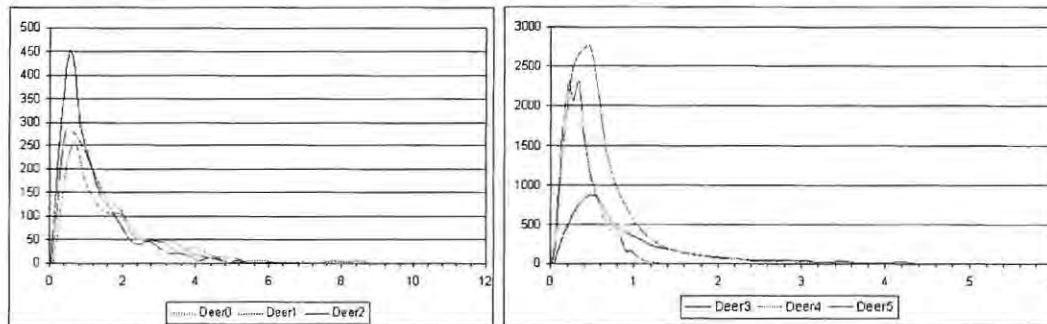


Figure 34 - Edge length distributions: a) Deer0-Deer2; b) Deer3-Deer5

Another look onto the same data is shown in Figure 35, where we graph accumulative percentages (lengths are given in total, but arbitrary units). All of these curves can be approximated with  $1 - c \cdot e^{-bx}$  type functions (in these cases with  $c=1.1$  and  $b$  between 0.7 and 1.4), which makes their shape fairly predictable. Since the constants  $c$  and  $b$  depend on object-shape, optimisations already performed on the object and many other variables, finding a general working formula to establish values for the constants is unlikely. This makes it difficult to determine a cut-off value that will for example discard 10% of the edges. Visual quality is another factor and is discussed later.

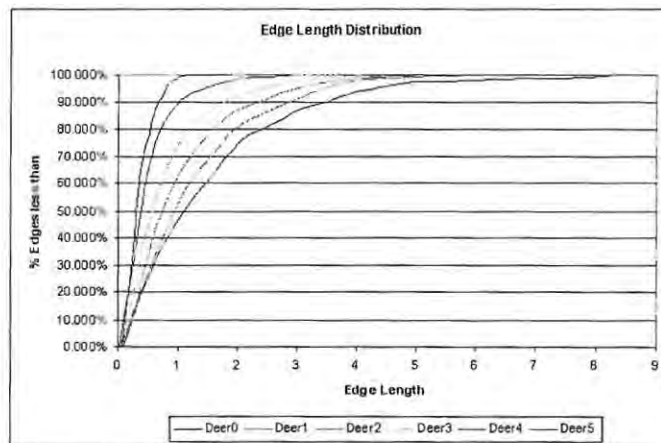


Figure 35 - Edge length distributions, accumulative percentages

The MaxAngle measure is somewhat easier to work with, because angles have well-defined bounding values (as opposed to the edge-lengths which are only bound by the longest edge, a somewhat recursive definition). Figure 36 shows two graphs, where we map number of edges against the angle between triangle normals of the triangles that define an edge. This is done because this measure starts with inface edges at angle 0 degrees, whereas the dihedral angle would begin at 180 degrees, otherwise they describe exactly the same thing. In Figure 36a) we can see that basically all edges are below 90 degrees. Figure 36b) is normalised in terms of total number of edges so that a comparison becomes easier. We can see clearly that only the very low resolution objects have edges above 90 degrees. Also the slope of the traces is basically linear (for all except the highest resolution object) up until about 25 degrees, which facilitates adjustment of the threshold value to a suitable value.

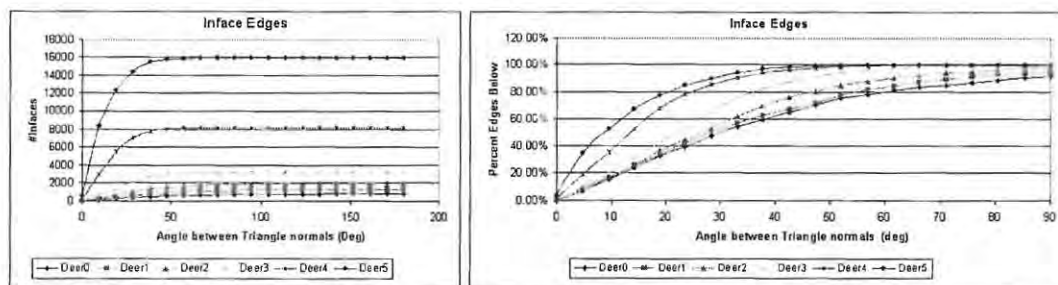












Figure 36 - Angle Distribution: a) Absolute Edges; b) Percentage of Total Edges

#### 4.5.1.2 Qualitative Results for Level of Detail Optimisations

While the results in Section 4.5.1.1 are interesting from a technical point of view, their practical implications in terms of visual quality of the renderer are of greater relevance. Table 14 shows the visual results for different values of MinLength and MaxAngle. We designed the test so that the largest threshold values would discard exactly 90% of the edges. Then we took 5 linearly interpolated samples (i.e. 0%, 25%, 50%, 75% and 100% of 90%). In the images in Table 14 we did not make use of hidden line removal in order to make sure that an edge that is not visible is non-existent (as opposed to hidden), which is the reason why even the first (supposedly perfect image) looks somewhat strange. Also the

reader will notice that the visual result for 0% is of course the same for both measures and mainly included for comparative reasons.

MinLen					
MaxAngle					
of 90%	0%	25%	50%	75%	100%
total	0%	22.5%	45%	67.5%	90%

**Table 14 - Visual Comparison of Edge-reduction approaches**

It is interesting to note that the two measures chosen by us seem to be fairly complementary: The MinLength measure discards object-detail with very high curvature, because to approximate such a curvature with straight line-segments, we need many short edges. Low-curvature object-detail on the other hand is well-preserved. In contrast, the MaxAngle measure discards low-curvature detail, because it assumes that such detail has less chance of fulfilling the silhouette criterion. Figure 37 shows this fact by combining both 100% samples, the MinLength version in red and the MaxAngle version in green. Coinciding edges are marked in black and are very few.



**Figure 37 - Combination of 100% MinLength and 100% MaxAngle**

This shows us that a combination approach should be employed when using these two measures and only edges that are discarded by both measures should really be eliminated. This ensures that the disadvantages of any single measure are counter-balanced by the benefits of the other.



### 4.5.2 "Unconnected Triangles" Optimisation

Owing to the fact that some edges are considered constantly unimportant due to their dihedral angle, we can implement another optimisation. We know that an edge is defined by exactly two triangles. Therefore, we can disregard any triangle that is not part of any important edge in the triangle-traversal, which determines edge-visibility. An interesting result here is that the number of connected triangles remains high even for a relatively low number of important edges, so that this optimisation is not very effective.

#### 4.5.2.1 Quantitative Results for "Unconnected Triangles" Optimisation

As already briefly mentioned in Section 4.5.1, we tried another optimisation, the unsuccessful result of which is so surprising, that we detail it in the following. We believed the underlying idea sound enough: As we discard edges through our two edge-reduction techniques, we are likely to be left with triangles that are no longer connected to any of the remaining edges. As these triangles do not have to be considered for face-orientation purposes (which in turn determines the silhouette condition for edges), we could theoretically reduce the computational load on the system quite considerably, by also discarding those unconnected triangles. The relevant graphs in practice are shown in Figure 38. The first trace represents the number of MaxAngle-edges as a percentage of the total number of edges for each of the different test-objects. The second trace shows the number of MinLength edges in a similar fashion. Both Max-Angle and MinLength criteria were chosen to produce a decent visual quality and were the same for all objects. The third trace shows the remaining edges that were left after edge-reduction. We can see that in this particular case the MaxAngle reduction had a much more predominant effect on the reduction procedure. The next trace shows the percentage of triangles that are still connected to remaining edges after edge-reduction. We can see for the first four objects that even though as much as 35% of edges are culled, the percentage of connected triangles remains as high as 93%. Only when the number of culled edges increases above 35%, do we detect an analogous decrease in connected triangles. The last trace shows the time spent on the triangle computations, compared to that without the optimisation, and aligns, according to expectations, with the Connected-Triangles-trace. We thus conclude that there is somewhat of a speed-up achieved by this optimisation, but it is not nearly as big as we hoped it would be.

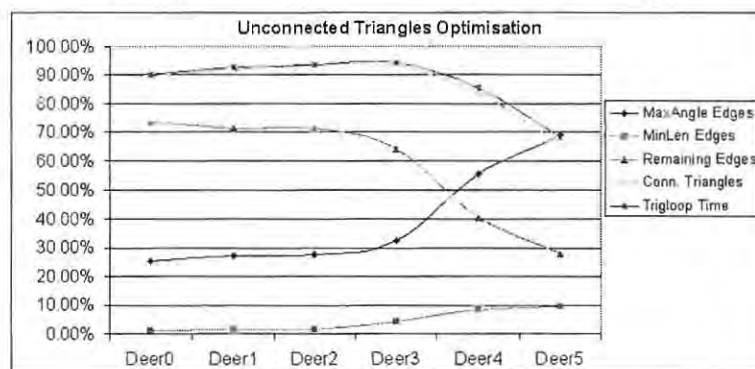


Figure 38 - Unconnected Triangles Graph

### 4.5.3 Recursive Algorithm Optimisations

Several issues need to be acknowledged:

- A recursive algorithm requires additional storage capacity and represents a computational overhead
- Comparing the length of segments requires many square-root operations

The first point, while being valid is not too significant with regards to storage capacity, as this can be considered bountiful (all systems on which this renderer is supposed to run have at least 64MB of user memory available). The issue of computational overhead (saving and restoring the recursive function stack as well as other related operations) would have to be compared with an equivalent modified linear algorithm. The second point of this list is easy to deal with. By stating the maximum length in squared form, we can get away with calculating only the squared lengths of the line segments.

Yet another optimisation we successfully experimented with is the use of displaylists to further rendering performance. Displaylists are sets of OpenGL instructions, compiled into a list (this involves conversion of instructions and data into OpenGL-native structures, thus greatly decreasing execution overhead). Usually displaylists are used in situations where geometry does not change (once a displaylist is compiled, its arguments are static. This means, that vertices, rotations, texture information etc. cannot be changed after the displaylist has been compiled). We investigated their use despite the fact that the geometry of our RPS renderer alters on two levels:

- The random perturbations themselves represent varying geometry
- Edges belonging to the silhouette change with object rotation

We address the first point by compiling a whole set of displaylists for a given view and object. This way we can call the displaylists in any given order and thus create the impression of randomness. Our implementation keeps a list of displaylists, allowing repetition, and randomises this list every  $n$  frames. A sequence of displaylist calls could thus look like  $\{3,2,2,1,3,1\}$ . The effect is that of a semi-random silhouette, very much like those of stop-motion animations in film and advertisements.

The second point can be exploited by recognising object rotation relative to the viewer and only updating displaylists when this situation occurs. As we mentioned in Section 2.2.3, we do not take the relative position of the viewer to the object into account, so that object translation will have no effect on the edges being considered silhouette. The same applies to scaling, so that of the object transitions to be encountered only rotation needs to be considered. If we were to update our displaylists on each frame when the object undergoes rotation, we would lose the benefit of using these displaylists, so we simply don't update them on each frame. For an explanation why we can get away with this, we consider the following. Figure 39 shows an arbitrary object that rotates relative to the viewer. Two views are shown in each frame: one from the side and one from the position of the viewer (front). The rotation is anti-clockwise in the side-view and upwards from the viewer's perspective. It is clearly evident which edges

are visible, which ones form the silhouette and which ones the object hides as it rotates: In frame a) both edges are visible, none is considered silhouette. In b) the green edge marks the silhouette, but both are still visible. The situation changes slightly in c), where both edges are on the silhouette, but only the red one should be visible. Finally, in c) the red edge represents the silhouette and the green one (faded) is hidden by the object.

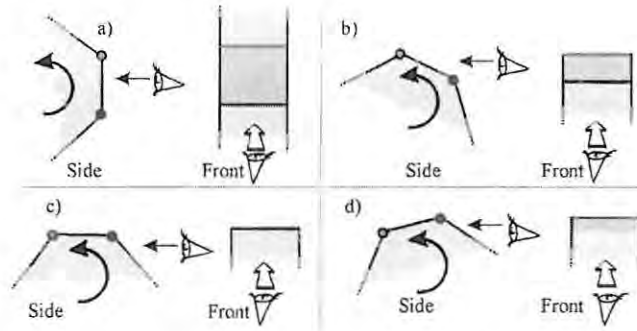


Figure 39 - Silhouette changes under rotation

Let us now consider the situation where we do not update the silhouette information for several frames, i.e. we keep the silhouette information of b) as we rotate the object through positions c) and d). The effect would be this: the green edge would be considered silhouette and therefore rendered (not so the red one). In case c) this does not matter as the two edges are in line with respect to the viewer. In case d) we would incorrectly render the green edge instead of the red one. The error this situation generates we call the *edge error*. We define the edge error as the distance between where an incorrect edge is rendered and where the correct edge is located (in our example the vertical distance between the green and the red edge). It is not entirely true that all incorrect edges can be considered as merely being at the wrong position; some may also be of different length or at a different angle, but for most smooth objects of reasonable tessellation this assumption is fairly accurate. Even for an extreme object like a cube (in the sense that angles are very large everywhere, making for potentially large edge errors), we can show the following: If we assume an average viewing distance of several cube-lengths and an update rate of 10 frames per full rotation (a fairly conservative estimate, as this translates to a speedy 36 degree rotation per frame), we generate relative edge errors of 5% (relative meaning in relation to the side-length of the cube). As 5% is roughly the uncertainty by which we perturb our edges, the edge error effect is not distinguishable from the random perturbations deliberately produced by the renderer. We have thus shown in theory why caching of edge-information can be used even under rotation. It is interesting to note that while this optimisation works very well on some systems, it fails to show considerable improvements on others. Our explanation for this is that displaylists, which we use as a caching mechanism, are not always equally optimised on all graphics cards or their software drivers.

With the above considerations in place, we devised the following algorithm for our random perturbation sketch renderer:

```

Let D = array of DisplayLists
Let dis = array to determine sequence in which D are used
Let Counter = indicates number of times Render has been called
Let Nx = Normal (at any of Vertex, Triangle, etc.)

```

```

Let View = View Vector

Frac(Edge)
  If Edge->Length < minLenth
    Draw Edge // Simple Line, no Clipping, no Culling,
              // no Blending, Depth Buffer
  Else
    Let Mid = MidPoint(Edge)
    Let E1 = Edge->Start..Mid
    Let E2 = Mid..Edge->End
    Jitter (E1->Start, E1->End, E2->Start, E2->End)
    Frac (E1)
    Frac (E2)
  End If
End Frac

GenerateAndDisplay(List)
  For each Triangle T do
    calculate Nr.View
  Next Triangle

  For each Edge E do
    If silhouette(E) // determined by above calculations
      Repeat between 1 and 4 times
        Frac(E)
      End Repeat
    End If
  Next Edge
End GenerateAndDisplay

Render
  // Render the Object into the Depth Buffer for HLR purposes
  call HLRDisplayList where HLRDisplayList is
  For each Triangle T do
    Draw T //fill mode, Texture(-), Fog(-), Color(-),
  Next Triangle //Culling(+), Blending(-), Color Mask(+),
              //Lighting(-)

  Every nth frame do
    Randomise (dis)
    Clear D
  End Every

  If justCleared(D[dis[counter]])
    GenerateAndDisplay(D[dis[counter]])
  End If

  call (D[dis[counter]])

  Counter++
End Render

```

Listing 6 - Random Perturbation Sketch Algorithm

Some examples of our random perturbation sketcher are displayed in Figure 40 a-d). The trained eye will be able to spot the Hidden Line Removal (HLR) artefacts discussed in Section 2.3.2 (HLR object may be cutting through object lines due to the random perturbations) in the stairs of the diving board, the crown of the chess piece and the head of the salesman. While these artefacts are extremely difficult to spot in still images, they are more noticeable under animation, because the artefacts – due to the static nature of the HLR object – tend to stay fixed, while the lines around them change appearance. The artefacts themselves are also rather typical for the locations they occur in: The steps of the diving board are made up of relatively fine geometry while the HLR object is quite heavily scaled which results in an incorrect displacement of the HLR object relative to the rendered strokes. Similar applies to the crown of the chess piece and the head of the salesman. As the HLR object is scaled towards the middle, they interfere slightly with the upper extremities of the rendered objects. Again, as these issues are only noticeable to the expecting eye, we argue that they can equally well be attributed to the sketching style itself.

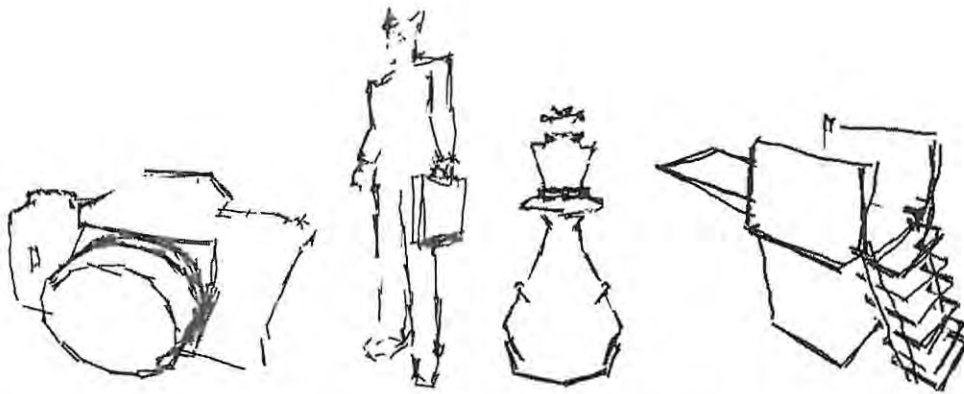


Figure 40 - RP Sketches: a) Camera; b) Salesman; c) Chess Piece; d) Diving Board

#### 4.5.4 Object-segmentation approach

Even though the recursive algorithm approach works well and its overall performance is very satisfying, one can observe a jolting behaviour whenever displaylists are updated, because performance drops drastically only to resume at full speed once the update is complete. While this is not visually unsettling (as the viewer expects randomness and low update-rates [which are different from frame-rate]) we investigated other options to achieve the same visual and temporal effects with a more constant frame-rate.

Our modified approach, even though producing the same visual result, is radically different from our initial approach. In essence we consider each edge not part of the object, but an object in its own right. The idea for this stems from the notion of a Matrix-stack in OpenGL (and other APIs). Depending on the relative size, position and orientation of different objects in a given scene, the Matrix-stack is used to group together related objects in order to synchronize their transformations. We thus perceive our single object as a group or cluster of related objects that move in unison. The way we achieve this is shown in Figure 41: We consider a given target edge and the unit vector on the x-axis (the latter is an arbitrary choice, any constant and well-defined vector would do). We then establish the scaling, rotation and translation (in that order) necessary to transform the unit vector into the desired target vector.



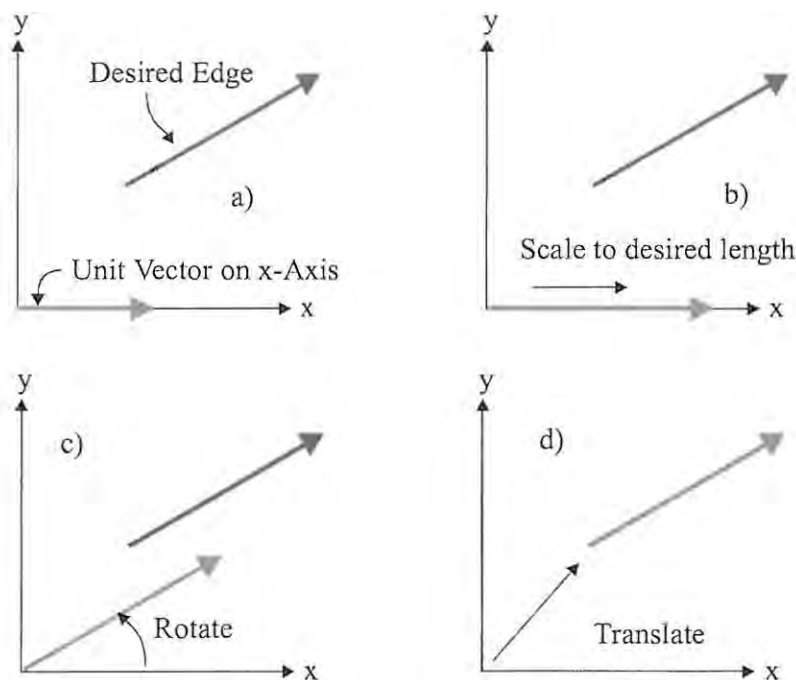


Figure 41 - Transforming Unit Vector to Desired Edge

Since any of the above-mentioned affine transformations can be represented using a 4x4 matrix, we can multiply the scaling, rotation and translation matrices together to obtain one single transformation matrix that will transform the x-axis vector into our desired vector. This allows us to render the important edges by transforming the x-axis unit vector into its correct position within the object and then drawing it. So far we have not really achieved anything we couldn't have done more straightforwardly. The crucial idea is that we are not limited to rendering the x-Axis vector and transforming it. Indeed, we can render anything in its place and perform the transformation just the same. In our case it makes sense to draw a *sketchy* line instead of the straight unit vector and that is exactly what we do. In fact, to optimise performance, we pre-render several sketch-lines all of which are obtained by applying some combination of the uncertainty-functions defined in Figure 33 and storing them in displaylists. When it comes to actually rendering an edge, we choose one of the displaylists according to a temporal-dependent semi-random value, transform it and render it. The workings of this approach are shown in Listing 7. This means that instead of applying the uncertainty functions at run-time, we apply them once off and re-use them from then on. It also means that we do not have to rely on inter-frame coherence to improve performance. In fact, the object-segmentation approach is so much faster than the recursive-algorithm approach, that we allowed ourselves to perform perspective correct view-vector calculations instead of the approximations discussed in Section 3.5.2.1.

```
Render
// Render the Object into the Depth Buffer for this purpose
call HLRDisplayList where HLRDisplayList is
  For each Triangle T do
    Draw T //Fill mode, Texture(-), Depth Buffer(-),
  Next Triangle //Culling(-), Blending(-), Color Mask(-),
               //Lighting(-)

  For each Triangle T do
    calculate Nr.View(T) // Notice View depends on T
```

```

Next Triangle
For each Edge E do
  If silhouette(E) // determined by above winProjections
    PushMatrixStack // push current matrix so we can restore it
    MultMatrixOntoStack M(E) // Apply Matrix of current Edge
    Draw Sketchy Line // Draw a sketchy line
    PopMatrixStack // restore previous matrix
  End If
Next Edge
End Render

```

### Listing 7 - Object-Segmentation Algorithm

There are some issues with using pre-cached uncertainty functions instead of calculating them at run-time. In Figure 42 we show a cached sketch line being stretched longitudinally by various factors. While the longest line seems almost straight, the shortest one looks a lot more perturbed, even though their perturbation amplitudes are all the same. This is due to the fact that the short line has a lot more segmentations per unit length than the longer one. If we imagine several very short edges close to one another (as is usually the case for regions of medium to high curvature), we'd get the result of extremely segmented lines, while regions with lower curvature will look almost non-perturbed. To apply a more homogenous appearance to the object, we create various bins, into which we sort edges according to their lengths. For each of the bins we create several display-lists. Short edges will have fewer segmentations than long ones and any given edge can only choose a displaylist from its allocated bin.



Figure 42 - Different longitudinal Stretch-factors for a Sketchy Line

Another related issue is also concerned with scaling, and that is the amplitude of the perturbation uncertainty function. All our pre-rendered sketchy lines are one unit long and are then scaled in the x-direction by the length of the edges they are supposed to represent. The perturbations also have some absolute amplitude on the unit scale, so they obviously also need to be scaled by some value before they are applied to the object (otherwise a large object would look very straight and unperturbed, while a very small object might be totally disconfigured by the perturbations). Our first approach was therefore to make the amplitude linearly dependent on the length of the edge, with good results, except when encountering very long edges (as appear in many optimised objects) which would be overly perturbed (this effect is all the more noticeable, since the perturbations themselves are animated). Experimenting with other relations than linear proved unsatisfying. Instead we found our solution in using a threshold-value that is dependent on the object's bounding box dimensions. Therefore, the perturbation amplitude of an edge is linearly proportional to its length, unless the thus computed amplitude exceeds a threshold value, in which case it is clamped to that threshold value. This approach produces visually pleasing results for any size object and is also easy to implement.

## 4.6 Extensions

### 4.6.1 Pencil and Coal Sketching

The previously discussed idea of pre-caching uncertainty information proved to be very effective and efficient. To vary the visual appeal, we searched for ways of applying a certain width and texture to our sketch strokes in order to create the illusion that the objects were drawn not only by hand but also with a special tool or medium, like e.g. a pencil, coloured chalk or coal. Without much difficulty we found the answer in the question: textures. By choosing and applying the correct textures, we could not only give texture to our strokes, but the textures themselves could produce all the randomness and imperfections of a typical hand drawn stroke as we exemplified in Figure 43.



Figure 43 - Several Stroke textures

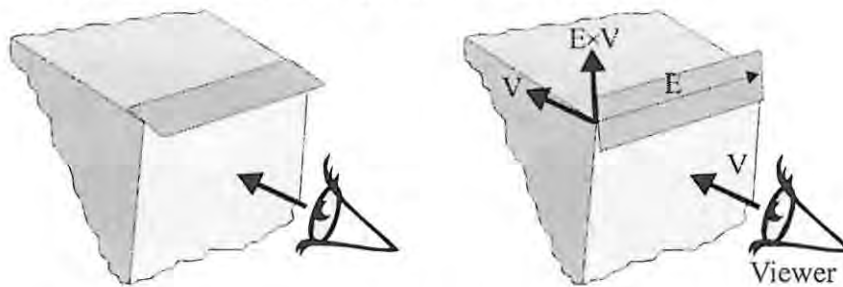
The challenges we were faced with by changing from sketchy-lines displaylists to textures are listed as follows:

- Assignment of textures to edges
- Orientation of edges
- Scaling of textures
- Appearance / Disappearance of Edges
- Blending / Co-existence of edges

We decided for this version that we wanted a more static look (as opposed to the constantly changing one of the random perturbation sketcher). To achieve this, we load a variety of stroke textures into the system and randomly assign one of these to each edge of the object. This assignment never changes during the lifetime of the renderer. If an animation-effect is wanted or needed, textures can easily be blended to change smoothly from one texture to the next.

The idea that an edge should be oriented in a certain way is actually a strange if not logically impossible one. The reason for this is that an edge by definition is just a line of zero width and just like a point that cannot actually point at anything (another tragic misnaming) an edge has no well-defined normal (as for example a plane has). Obviously we cannot apply a texture to a zero-width edge, so we have to widen each edge artificially. To achieve a homogenous result, we use the same method that we employed to scale the perturbation amplitude. After all, the amplitude of perturbations is in this case limited by the width of the applied texture and the two are therefore equivalent. The width of the extension is thus proportional to the length of the edge limited by some cut-off value. The next question that arises now that we have given width to our lines is how to orientate them. If we assign arbitrary orientations to them and leave them viewer-independent the result is a confusion of edges which change width as the object

undergoes rotation relative to the viewer (this is due to the fact that the viewer might look onto the edge of an extended edge and thus again see nothing, as illustrated in Figure 44a)).



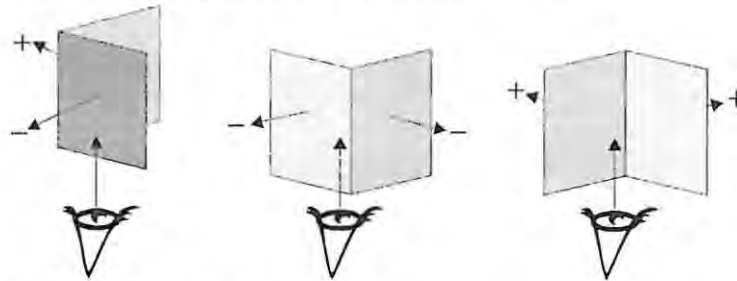
**Figure 44 - Edge extensions: a) static solution; b) view-dependent solution**

Our solution uses view-dependent orientation of edges and is demonstrated in Figure 44b). If the view-vector is  $V$  and the vector running alongside the edge is called  $E$ , we can use the cross product of these two vectors to find a vector which is perpendicular to both of them (if both  $E$  and  $V$  are well-behaved, i.e. neither of them is zero and they are neither parallel nor anti-parallel – in these cases the cross-product would yield the zero vector, which by definition is perpendicular to all other vectors, thus still complying with our reasoning, but not being useful as an extension vector). This means that the exposure towards the viewer is maximal (and is probably the reason why a slight variation of this technique is called *Billboarding* – because in advertising the product has to be maximally exposed to the customer). The special case mentioned above occurs when the viewer looks directly onto (in the direction of) an edge. It can be argued that such an edge would not be visible anyway. An edge approaching such a condition in fact turns smaller and then disappears when the condition is met exactly. In practice we call this a feature rather than a problem, because the visual effect is very smooth and deals nicely with these special edges by shrinking them smoothly until they vanish, thus avoiding noticeable popping (sudden appearance or disappearance of scene elements).

Next we had to deal with the issue of different length edges just like we did with the segmentation approach. In this case it is not the pre-cached sketchy line that is either stretched or compressed, but the texture itself, but the visual problem is exactly the same. Our solution on the other hand is slightly different in this case. We define a measure dependent on the size of the object and scale our textures according to this while allowing texture repetition to occur. This means that short edges might not be mapped with a complete texture but only part thereof, while long edges may repeat a given texture several times for a complete fill. This solution is easily implemented and produces pleasing and realistic results (care has to be taken when generating or designing the textures so that tiling/repetition of textures cannot be noticed easily). Seeing as the lengths of edges do not change, the texture-coordinates for each edge can be stored as part of the edge information and recalled when necessary. This alleviates the need to recalculate the length and texture-coordinates of an edge at run-time.

Another problem that we were faced with was that of edge popping caused by varying edges fulfilling the edge-criterion. This criterion is either fulfilled (an edge is visible) or not (an edge is not visible) and since the change for the condition is instant, edges may pop in or out of view and in extreme cases even flicker.

For an animated renderer like the previous line-renderer where a lot of noise is generated in the image this is not easily noticeable, because the randomness of the animation conceals the edge-popping. For a non-animated renderer on the other hand this kind of effect is extremely easy to spot and also visually highly unpleasant. Our solution is as simple as it is effective: We fade in edges that are about to become visible and likewise fade out edges that are about to disappear. The obvious question is how do we know when an edge is about to appear or disappear. The measure we use to determine this is the product of the dot products between the normals of edge-adjacent faces and the view-vector.



**Figure 45 - The three possible edge configurations: a) silhouette edge; b) front facing; c) back facing**

The dot products are available anyway, since we use them to determine the silhouette condition. Figure 45 shows the three possible edge configurations: a) one of the edge-adjacent faces points toward the viewer, the other one points away (negative and positive dot product respectively). The product of the two dot products is necessarily negative. b) both faces are front facing. The combined dot product is positive (because both dot products are negative). c) both faces are back facing. The result is similarly positive, since both dot products are positive. With this in mind we get a curve similar to the red, solid cosine in Figure 46 (In general the product of the dot products will not be a cosine and depend on the angle between the two faces. We use the cosine function as a first approximation to illustrate our approach which will work with any function exhibiting the above-mentioned features). Even though we cannot be sure if the positive regions indicate front or back facing faces (indicated by the question marks), we know that the silhouette condition is true for a negative result. We therefore construct a function that rises rapidly as the cosine-like function approaches positive zero, reaches 1.0 (full visibility) at zero and remains 1.0 for any negative value. This user-defined function need not rise linearly and the rising-point is also arbitrary. In fact we chose to implement the function in form of a look-up table to optimise speed and flexibility in function definition. The symmetry of the function shown in Figure 46 is automatically guaranteed, because the inverse cosine is only defined in the interval  $[0 \dots \pi]$  (N.B. the graph shown uses  $x$  units of  $\pi$ ). Of course the visibility-function defined here is a prime example of an applied importance-function as defined above.



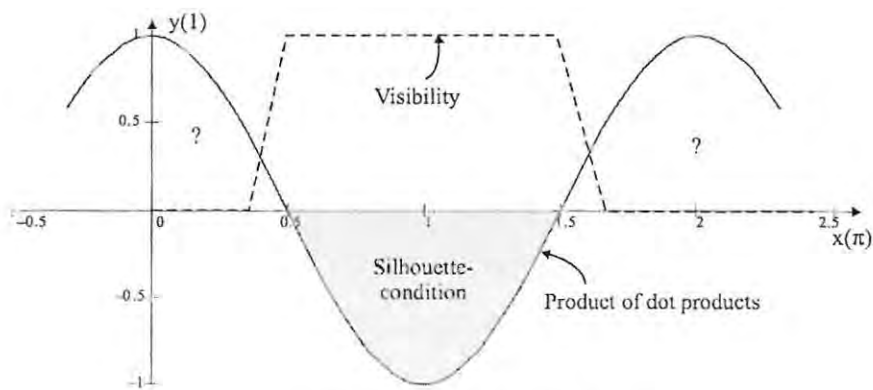


Figure 46 - Fading function for edges

In our implementation we chose the rising value so that an edge fades in and out over a five-degree span. The result is a total elimination of edge popping and a visually very smooth appearance (see Animation D for a demonstration). Another interesting and enriching side effect is that faded edges are rendered much fainter than fully visible edges. Visually it appears as if the person drawing the sketch was uncertain about a specific line and rather just hinted at it (see Figure 47a through d for examples).

A depth-buffer related problem arises when actually rendering the edges. Extended edges that converge in a single vertex (like at the corner of a box) are almost guaranteed to overlap from the vantage point of the viewer. If normal depth-testing and writing were enabled, the first edge to be rendered would write itself into the depth buffer and thus prevent other edges from rendering to the shared pixels. This results in the ends of edges being angled and cut-off abruptly – visually very displeasing. Our solution here is to keep using the depth-test for hidden line removal purposes, but to disable depth-buffer writes with `glDepthMask (GL_FALSE);`. This allows edges to render over each other, which is not unrealistic, as the same would happen if strokes were produced manually. For the above-mentioned method to work, the stroke textures have to represent or contain an alpha channel, which prevents pixels from being generated where the texture is white (no stroke). A separate alpha-channel in the frame-buffer on the other hand is not necessary.

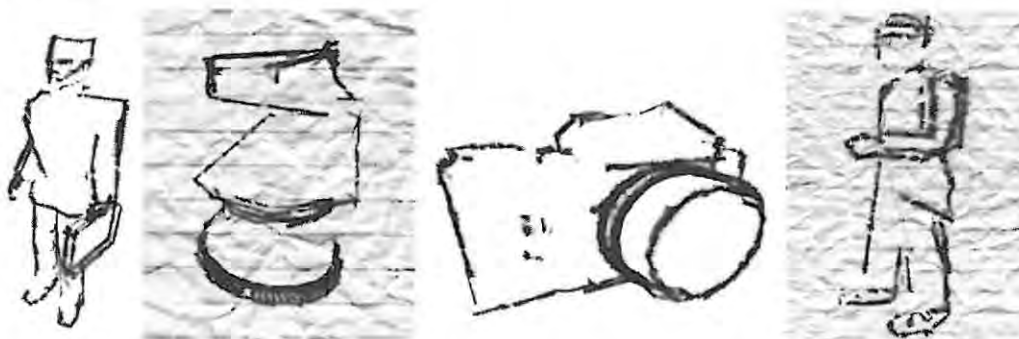


Figure 47 - Charcoal sketches: a) Salesman; b) Chess Piece; c) Camera; d) Walker

As we show in Listing 8 the algorithm for our coal sketch renderer is very simple in design. In practice it is also highly efficient and outperforms the recursive-algorithm sketch renderer by a wide margin (see Section 4.7.1 for details).

```

Let L = Light Vector
Let  $N_x$  = Normal (at any of Vertex, Triangle, etc.)
Let View = View Vector
Let Q = Billboarded Quadrilateral

Render
  // Render the Object into the Depth Buffer for HLR purpose
  call HLRDisplayList where HLRDisplayList is
    For each Triangle T do
      Draw T //fill mode, Texture(-), Depth Buffer(+),
    Next Triangle //Culling(+), Blending(-), Color Mask(+),
                  //Lighting(-)

  For each Triangle T do
    calculate  $N_x \cdot \text{View}$ 
  Next Triangle

  For each Edge E do
    If silhouette(E) // determined by Area dot product
      Q = billboard(E)
      Draw Q // fill mode, Texture(+), Depth Buffer(+),
    End If // Depth Write(-), Lighted(-), Culling(-),
  Next Edge // Blending(+)
```

End Render

#### Listing 8 - Algorithm for Coal Sketching

In order to increase the variety of coal textures, other than physically creating more, we can combine several existing textures via multi-texturing or blending. Another related effect can be achieved by blending between different textures, thus allowing edges to change their texture at run-time while the object remains at a more or less constant brightness level (i.e. is not flashing wildly each time the textures are altered). In addition to this the colour and/or brightness of the textured quads can be varied to create even more render variety by easy means.

#### 4.6.2 Sketch shading

As already mentioned in Section 4.2.1 there might be cases where shading is wanted or required to add texture, form or depth to an object. So far we have only addressed rendering of edges and so we will now discuss our method of shading objects with sketchy lines.



Figure 48 - Various hatching suggestions ([24])

Cross-hatching (or hatching for short) is a technique which uses roughly parallel strokes and/or crossed strokes to indicate shading (see Figure 48 for examples). Most of the time the strokes do not follow object geometry (but exceptions do exist, see Hall [28] and Praun et al [68]). This is due to the fact that a real sketch is usually drawn on flat paper and it is easier to perform strokes in more or less the same general direction. Additionally, the strokes themselves indicate shading thus indirectly hinting at the underlying object shape. Following object geometry would therefore be redundant and require special effort. Consequently, it is mainly done for stylistic purposes. Section 2.1.3 already introduced one possible technique using geometry sub-division and a set of hatching textures (Lake et al, [42]). We demonstrate another approach not requiring sub-division of geometry, which can be extended in a multi-pass scheme to allow the use of multiple simultaneous hatching textures. The individual issues we have to deal with are:

- Generation of planar hatching strokes
- Conveying shading-information with hatching strokes
- Multiple hatching textures and use of the comic algorithm

To ensure that hatching strokes are planar with respect to an imaginary image or paper plane, we use a modified version of a more general projective texturing approach found in [50], Node 49. Without going into unnecessary detail, the method works like this: OpenGL provides functionality to automatically generate texture co-ordinates for a given geometric vertex. The generation-functions have to be enabled individually for each texture co-ordinate to be generated. The functions themselves can be specified in the form of several default mapping functions or, as we did, the user can specify them directly in the form of a matrix representing affine transformations. The mapping that is required for planar projection onto the view plane from the eye-point of the viewer is identical to the mapping required to project the geometry (a schematic for this is shown in Figure 49). The matrix used is therefore the identity matrix. The following code shows the necessary steps to perform this projective texturing:

```
static GLfloat Splane[] = {1.0f, 0.f, 0.f, 0.f}; // this defines
static GLfloat Tplane[] = {0.f, 1.0f, 0.f, 0.f}; // the unit mapping
static GLfloat Rplane[] = {0.f, 0.f, 1.0f, 0.f}; // matrix
static GLfloat Qplane[] = {0.f, 0.f, 0.f, 1.0f};

// enable texture co-ordinates generation functions
glTexGenfv (GL_S, GL_EYE_PLANE, Splane);
glTexGenfv (GL_T, GL_EYE_PLANE, Tplane);
glTexGenfv (GL_R, GL_EYE_PLANE, Rplane);
glTexGenfv (GL_Q, GL_EYE_PLANE, Qplane);

glEnable(GL_TEXTURE_2D); // enable 2D texturing

// Enable texture co-ordinate generation functions
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_R);

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

GLfloat mvm [16], pm[16];
glGetFloatv(GL_MODELVIEW_MATRIX,mvm); // get Modelview matrix
glGetFloatv(GL_PROJECTION_MATRIX,pm); // get projection matrix
glMatrixMode(GL_TEXTURE); // change active matrix stack
glPushMatrix(); // save current state
glLoadIdentity(); // reset state
```

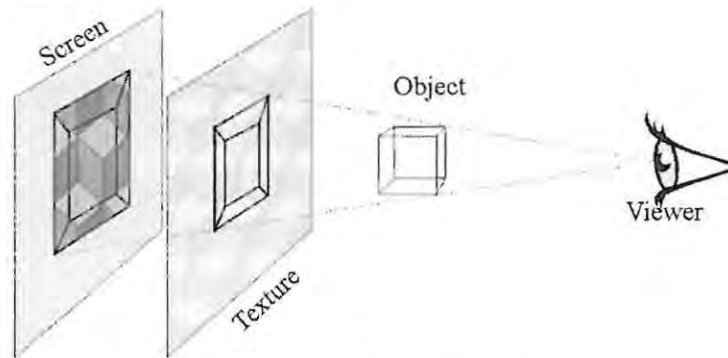
```

glRotatef(tex_rotation_angle, 0.0f, 0.0f, 1.0f); // rotate texture
glScalef(tex_repeat_factor, tex_repeat_factor, 1.f); // scale texture
glMultMatrixf(pm); // set perspective transform (from viewer)
glMultMatrixf(mvm); // set model transform
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glMatrixMode(GL_MODELVIEW_MATRIX); // change active matrix stack

```

**Listing 9 - Using projective texturing**

Apart from specifying the current perspective and model-view matrices - to be used to transform texture co-ordinates - by copying them from the respective matrix stacks, several other operations can be performed to change the appearance of the hatching strokes. Firstly, the texture representing the strokes can be rotated to achieve angled strokes and secondly, we can scale the texture to repeat it more or less often across the screen.



**Figure 49 - Projective Texturing**

If we now simply went ahead and applied the projective texture to the whole object indiscriminately, the whole object would be textured and we would not produce the desired light-dependent shading effect. We therefore use transparency to determine where hatching strokes will be visible and where not. Our first approach was to vary the transparency at each vertex according to the  $N_v \cdot \text{Light}$  product (which we previously used in our Comic renderers). As this produces diffuse shading on the object, the resulting images were too smooth and unconvincing, so we applied some thresholding and clamping in order to enhance the contrast. The visual result and performance of this approach are excellent at least for static scenes (see Figure 53a) through d)). When the object rotated relative to the light-source on the other hand, the thresholding becomes disturbingly obvious. Figure 50 shows images from a scene, where the light was rotated every so slightly from one image to the next. We can see that the vertices marked by triangles all exhibit constant transparency (all 0% except the right-most one, which is 100%).



**Figure 50 - Per-vertex Transparency Shading**

The vertices marked by circles on the other hand change their transparencies instantaneously. Since the transparency values for the triangles defined by the vertices are interpolated linearly, we get an *all-or-nothing* effect which presents itself in sudden appearance or disappearance of visible regions (usually referred to as *popping* or *clicking*). It is evident that this effect's pronunciation is inversely proportional to the density of the object tessellation, but we want our algorithms to work with any kind of object (not just finely tessellated ones) and must therefore address this problem.

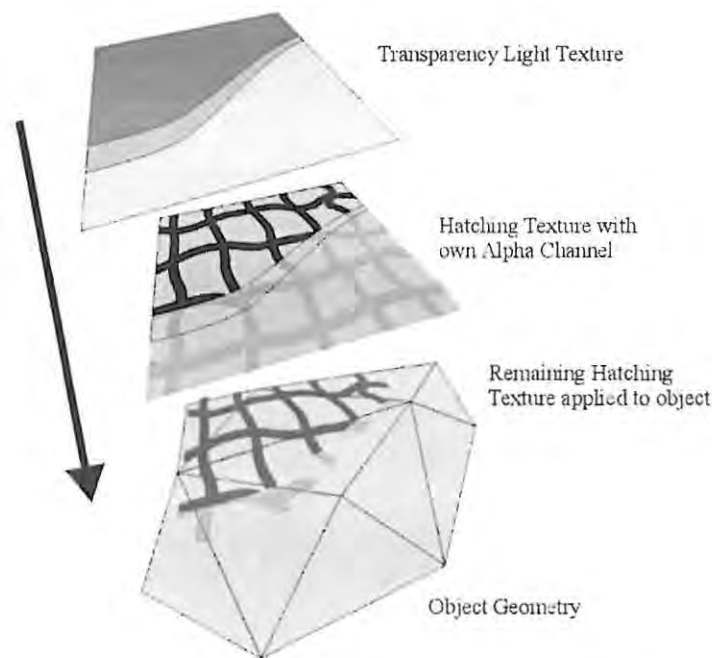
Lake et al [42] solve this problem by sub-dividing polygons until all vertices of a given polygon are in the same shade region. This also allows different hatching textures to be applied to different shade-regions but is computationally extremely expensive.

Our solution is related to our previously discussed Comic renderers. Using shading-textures, we are able to apply crisp, continuous shade-regions to our objects. We use the same principle here, as illustrated in Figure 51. Firstly, we use a one dimensional shade-texture (similar to the one we introduced for the standard comic renderer, just with a softer gradient between the principal regions) as a transparency-component (instead of brightness). Just as in the comic renderer, it is indexed by the  $N_v \cdot \text{Light}$  dot-product. This map is then combined with the hatching-texture, which itself also contains an alpha-channel (in order to mask out stroke-free regions). The combined textures are then applied to the object-geometry and appear as cross-hatch shading in the image. The result is very smooth, because texture co-ordinates are interpolated instead of direct transparency values, which allows for sharply defined shading contours, without the heavy computational overhead of triangle sub-division. Animation E shows this effect, as well as that of projective texturing, as the viewer moves relative to the object.

This method can be used with or without multi-texturing. If no multi-texturing is available, the different texture layers have to be applied in multiple rendering passes, which obviously increases the load on the graphics card immensely. In addition to that, a separate alpha-buffer has to be available to store the intermediate transparency values (this is not the case when using multi-texturing, because the textures are combined internally in the graphics card texture-units, before being applied to the object).

Applying different hatching textures to different shade-regions is also easily implemented with our approach. The reason one might want to do this is to simulate brightness with stroke-density. For this to look plausible, regions with fewer strokes (appearing lighter) have to be sub-sets of regions with more strokes (appearing darker) so as not to produce discontinuities. We then have to use a multi-pass approach: The first pass initialises the alpha buffer and subsequent passes then render the different shading regions with different cut-off values for the alpha-comparison-test (used by OpenGL to determine whether a pixel is drawn or not).





**Figure 51 - Hatching-shading using Multi-texturing**

Alternatively we can use a 3D texture (the logical extension of a 2D texture) as shown in Figure 52. If we call the extra dimension *depth*, then the *width* and *height* of such a texture would represent our standard hatching-texture. Along the depth-axis we vary not only transparency, but also the hatching-textures used, so where we have less transparency, we also have less strokes (The first two textures in Figure 52 have 9 strokes, the next two have 5, the following has 3 and the remaining three none). Since 3D textures may be linearly interpolated on any axis, we can not only achieve a sharp and continuous transparency gradient, but also a smooth variation between darker and lighter versions of the hatching texture (see Animation F for a demonstration of an angled plane rotating under a light-source using this method). The only disadvantage to this method is the huge amount of texture-memory required<sup>1</sup>, but apart from that, we are able to apply banded hatch-shading with multiple hatch-textures in one single rendering pass and without multi-texturing (i.e. only using a single texture).

<sup>1</sup> For one normal quality hatching 3D texture, we need (height · width · depth · components)

$128 \cdot 128 \cdot 32 \cdot 4 \text{ bytes} = 2097152 \text{ bytes} = 2\text{MB}$

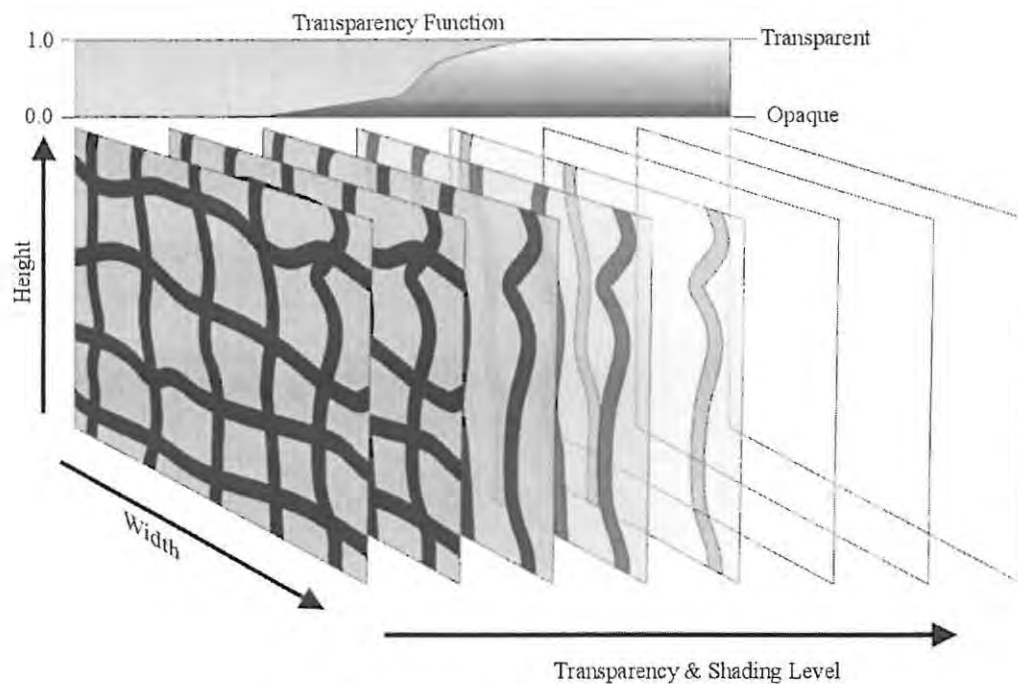


Figure 52 - Using a 3D texture for sketch shading

Regardless of which method we choose for a given implementation, the basic algorithm is shown in Listing 10, where the ellipses represent other pieces of code responsible for rendering important edges (depending on which kind of edge-rendering we perform this might be done before or after the shading stage and is mainly influenced by the order in which we want the z-Buffer to be initialised).

```
Render
    ... // other rendering instructions

    // render hatching shading
    For each Vertex V do
        calculate N_v.Light // dot product
    Next Vertex

    Set_Texture_Generation_Mode

    For each Triangle T do
        Draw_Transparency (T) // set transparency according to dot product
        Draw T                // Fill Mode, {Projective} Texturing(+),
                             // Depth Buffer(+), Lighting(+), Culling(+),
                             // Blending(+)
    ... // other rendering instructions
    Next Triangle

End Render
```

Listing 10 - Generic Hatch-Shading algorithm

Figure 53a) through d) shows the large variety of effects that can be achieved by simply varying the hatching texture and/or blending mode. The Skeleton in Figure 53a) is created with a white and black texture, the white creating a waxy-undercoat-effect, especially with a custom background texture. Figure 53c) in comparison uses a pure alpha map so that the background remains totally intact where no strokes are rendered. The statue of liberty uses another texture (this time truly cross-hatched), which was scanned

from a manual pencil hatching. Smooth and relatively regular strokes were used to create yet another look on the dog in Figure 53d)



Figure 53 - Cross-Hatch Shading: a) Skeleton; b) Statue of Liberty; c) Mouse; d) Dog

#### 4.6.2.1 Quantitative Results for Sketch Shading

In Section 4.6.2 we introduced several alternative methods to achieve the same hatching style and discussed how these methods affect the visual quality of the rendered object. To round this discussion off, we need to expand on the relevant performance results. Figure 54 shows traces for both the vertex-transparency solution and the comic-style multi-texturing solution. We can see that while the vertex-transparency approach performs slightly better throughout, the alternative multi-texturing approach does not lag far behind. For both solutions rates between 30 and 270 frames per second are achieved.

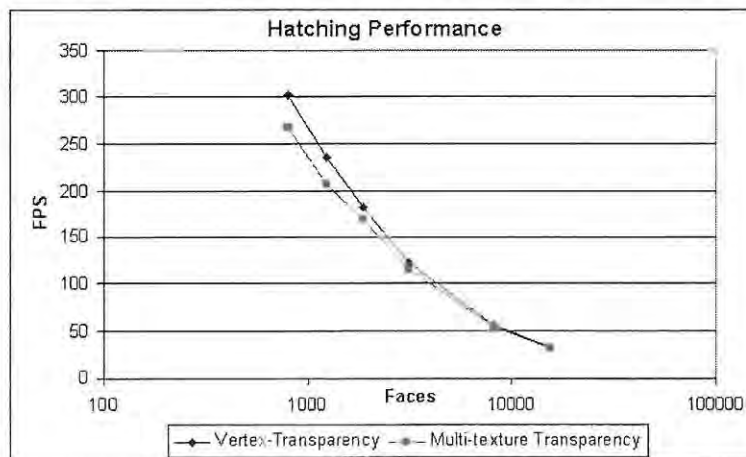


Figure 54 - Performance Results for Hatching

Regrettably, performance results for the 3D texturing solution (which technically is by far the most elegant) could not be obtained, because no graphics card in our laboratory would support 3D texturing in hardware (see Section 7.3 regarding availability issues) and software emulation is forbiddingly slow.

## 4.7 Results

### 4.7.1 Comparison of Approaches

In terms of absolute performance, the relevant graphs are shown in Figure 55. For these tests, we deliberately use no level-of-detail reduction, so that all edges in the object are initially considered important. Figure 55a) shows the results in terms of the total number of edges per object: The object-segmentation approach is clear overall favourite with the texturing approach coming second for less than about 5500 edges, while the recursive-algorithm approach outperforms the texturing approach beyond that mark. Results for the hatching sketcher are not repeated here, because we consider hatching a complementary style as opposed to a rivalling one. In addition, the primary object primitives forming the basis of the hatching style are different from those discussed here (faces instead of edges).

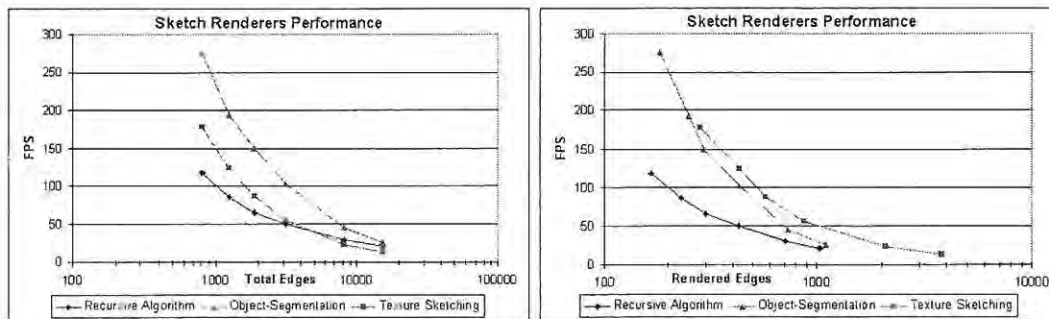


Figure 55 - Sketch Renderers Performance: a) Total Edges; b) Rendered Edges

While the first graph correctly shows the absolute performance of the renderers for a given number of edges, it does not truly represent their internal performance. What we mean by this is that each of the renderers actually draws a different number of edges. The reason for this is obvious when we look at the texturing sketch renderer: Here we fade edges in and out so that we end up rendering a large number of additional edges to that of the other renderers. The object-segmentation renderer on the other hand renders more edges than the recursive algorithm renderer, because it uses per-vertex view-vector calculations. Even though the difference in number of edges is fairly small in the latter case (under 10%), the correct view-vector calculation also incurs a higher computational overhead. It should be noted that the actual number of rendered edges depends on the current relative position and orientation of the viewer to the object and is said to be view-dependent. Thus taking into account the number of edges actually rendered (i.e. using the same y-values as before, but different x-values resulting in a horizontal shift of traces), we arrive at Figure 55b) and the result is notably different. We can see that the per-rendered-edge performance of the texturing sketch renderer is actually highest (except for the lowest object detail), whereas the relation between the two other approaches stays basically the same. That means that while the texture sketcher renders each edge faster than the other approaches, it has to render a lot more of them and as a result only performs second-best overall.

In this context and that of level-of-detail optimisations, it is quite interesting to have a look at the number of edges that are actually rendered. Below, Figure 56 shows such a graph for a typical view-point. Even though we did not perform any level-of-detail reduction on the objects, we observe a steady decrease of rendered edges in relation to the total number of edges. This can be explained as follows: As we increase the number of triangles per object, we also increase the total length of edges (i.e. the sum of the lengths of all edges). To visualise this we can image a triangle that is dissected along an arbitrary axis to form two triangles. In addition to the length of the edges of the original triangle, we now have to add in the dissecting edge. So while the total length of edges increases steadily, the length of edges describing the silhouette does not. Of course it does change slightly, but here the effect is somewhat different: Seeing as the silhouette is ideally a continuous curve, the effect of subdivision is that of dissecting the curve into smaller parts, which, when added up form approximately the same sum as fewer, longer segments.

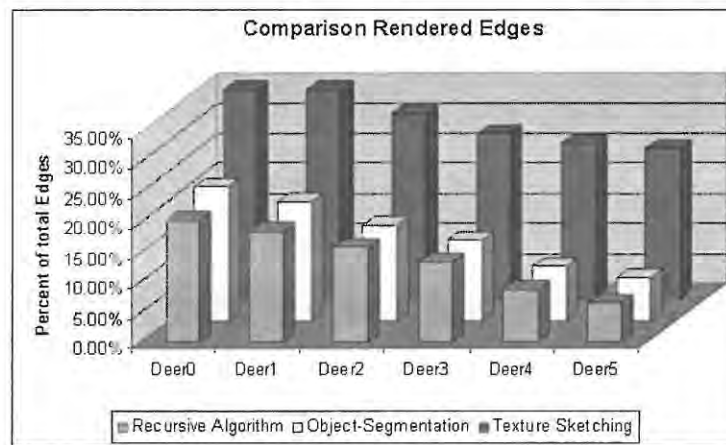


Figure 56 - Rendered Edges vs. Total Edges

Since the percentage of rendered edges is so low (maximally 35% for the lowest resolution object rendered by the texturing approach) and can even be as little as 7%, one might make the assumption that the images in Table 14 should look a lot better than they do. After all if we can discard 93% of all edges and still get a perfect image, why do the screenshots in Table 14 look so discontinuous when only discarding a maximum of 90%? The answer is simple: because the plot in Figure 56 is view-dependent. For a given view onto the object, we might only see 7% of the total number of edges and for another view this number might be very similar, but the edges that comprise those 7% are likely to be totally different ones. So while we might only see about 10% of edges at any given time, we are likely to see almost all edges during a full spherical rotation of the object. In practice we found that for our test-objects we can cull between 10-25% of the edges without impacting greatly on the visual quality.

## 4.8 Summary

We began this Chapter by listing the defining characteristics of manual sketch art:

- Drawn by hand (Randomness / Uncertainty-Factor)
- *Economy of line* (little, but important object detail)
- Few colours used (monochrome)



In order to reproduce these qualities with computer graphics, we identified the following problem-areas that needed solving:

- Convey a *manually-crafted* look (Uncertainty)
- Identify and apply important object detail semi-automatically (Importance)
- Render the specific object-detail with:
  - Deliberate Imperfections (Uncertainty, Randomness)
  - Rudimentary hints at shading if necessary (Monochrome)

From this list we then derived two significant concepts that help us formalise our discussion of sketch renderers: *uncertainty*-functions and *importance*-functions. The latter allows us to discuss a more generic sketch-algorithm without limiting ourselves by specifying explicitly which edges have to be rendered and which not. For our implementations however, we define the following concrete importance-functions:

- Static importance-functions:
  - MinLength edges (edges below a certain length)
  - MaxAngle edges (edges whose dihedral angle is above a certain threshold value)
  - User-defined edges (edges that the user deems important and flags as such)
- Dynamic importance-functions:
  - Edges that fulfil the silhouette-condition as specified in Section 2.2.3
  - Edges that are hidden by the object's body

By analysing hand-drawn sketches, we came up with a set of uncertainty-functions, which in combination and through the variation of their input-variables produce a large variety of sketchy lines (from architecturally precise to artistically chaotic). The main functions we identified and used in our implementations are:

- Repetition (of edge lines)
- Perturbation (of edges from their true geometry)
- Offset (of starting and end-points from their true geometry)

In our first approach, we showed how a recursive sub-division algorithm can be used to implement all the desired uncertainty-functions. An interesting contribution with regard to this is that we show that edge-information only needs to be updated every couple of frames without degrading visual quality in an animated renderer like ours. The reason for this is that the animated uncertainty-functions used by our renderer produce enough noise to drown edge-errors.

Our next insight was that dynamic computation of uncertainty-functions is computationally expensive and that static sampling and caching of these uncertainty-functions can prove very beneficial for the performance of our renderers. To this end we developed two renderers implementing this new approach.

The first approach, we call *object-segmentation*-approach and it uses displaylists of line-strokes to store the results of the uncertainty-functions. In order to be able to make use of displaylists for individual

edges, we have to consider each edge as an individual object, hence the name of the approach. This novel method of sketching seems counterintuitive at first (due to the breaking-apart of the original object structure) but redeems itself through its excellent performance and visual quality.

Our second approach stores uncertainty-information in the form of stroke-textures. We show how standard edges can be extended and oriented using a billboard-like-technique to maximise exposure towards the viewer and thus present a surface for the strokes to be mapped unto. We solved the problem of edge-popping (sudden appearance and disappearance of edges) in an easy to implement and resourceful fashion by fading them in and out. Our method for this can be applied without knowledge of when an edge is about to pop and is basically an extension of the edge-condition computation. Also the fading-function is completely customisable in shape (linear, exponential, etc.) as well as extent (how slowly or quickly edges are faded).

A common issue to these two approaches, regarding the *perturbation* uncertainty-function was discussed and successfully addressed. We discovered that for a visually pleasing look that is optically independent on object-dimension, we must scale the random perturbations proportional to the edge-length of a given edge, unless it is longer than a threshold value, determined by the bounding-box of the object. If it is longer, then the threshold value is used to scale the perturbation.

Next, we explained how cross-hatching can be implemented using projective texturing to allow for sketch-style shading. One of our approaches is extremely easy to implement, by thresholding alpha-values, but proves to be visually displeasing under relative rotation of object and light. Our second approach recycles the concepts of the standard comic renderer to produce heavily banded shading. In this instance, we apply it to perform blending instead of brightness-variations without the need for expensive surface-subdivision. We illustrate, how we can achieve appropriate blending without an alpha-channel and single-pass cross-hatching by using multi-texturing. We also discuss how multiple cross-hatch textures can be employed in a multi-pass scheme. Finally, we show how the novel concept of 3D textures is ideally suited to apply multiple cross-hatch textures with a customisable transparency-function in a single rendering pass. Our hatching solutions all perform between 30 and 270 frames per second (3D texturing performance could not be verified, due to lacking hardware support), rendering them suitable for use in connection with other NPR styles (i.e. in combination with the outlining sketchers).

In Section 4.5.1.2, we show what visual effects our edge-reduction schemes produce and discover that the combined employment of both schemes can guarantee a high degree of visual detail while reducing unimportant edges as much as possible. In terms of performance, we prove that all our renderers can perform between 20-275 frames per second for any of the test-objects, thus performing well within real-time constraints.

## 5 Painting

### 5.1 Introduction

#### 5.1.1 Definition

The enormous amount of radically different artwork that can be classified as *Paintings*, makes it extremely difficult to define the notion of a *Painting* as such. If we start with the verb *to paint*, we might be able to define a common ground onto which to build progressively. [30] defines to paint as : “to coat or decorate with paint”, or “to make a picture or portrait by using paint(s)”. It is obvious that the first definition could be used in the context of painting (the walls of) a house, while the second one describes the actions of an artist on some form of canvas. In actual fact and apart from the notion of artistic intent, the two are actually very similar and a rough approximation of factors to consider could thus look like this:

<b>Produced by hand</b>	<ul style="list-style-type: none"> <li>• Deliberate spatial (and temporal) flaws</li> </ul>
<b>Brushes</b>	<ul style="list-style-type: none"> <li>• <i>Shape</i></li> <li>• <i>Size</i></li> </ul>
<b>Paints/Colours</b>	<ul style="list-style-type: none"> <li>• <i>Palette</i></li> <li>• <i>Consistency</i> (Watery, Oily, ... )               <ul style="list-style-type: none"> <li>▪ Transparent/Opaque</li> <li>▪ Blending/Mixing</li> </ul> </li> </ul>
<b>Technique</b>	<ul style="list-style-type: none"> <li>• <i>Choice of {Brush, Colour, Strokes}</i></li> <li>• <i>Strokes</i> <ul style="list-style-type: none"> <li>▪ Direction, Length, Pressure</li> <li>▪ Number</li> <li>▪ Distribution</li> </ul> </li> </ul>

Table 15 – First Approximation of Painterly Factors

While it would be extremely difficult to define the above variables to such an extent that we can accurately model and replicate the personal style of a given artist, we assume they suffice in order to produce images that look *painted*. The visual output of our approach will validate this assumption. It should also be noted that it is not our intention to model physically realistic brushes or paints.

## 5.2 Problems

### 5.2.1 Problem Statement

In accordance with the items listed in Section 5.1.1 we must be able to develop simple but efficient models to simulate:

- Brushes
- Paints

to such an extent that will allow us to produce a variety of different styles and looks.

Furthermore, we have to develop an algorithm that describes the *artistic technique* to be applied. While we do not attempt a rigorous analysis of existing styles and their reproduction, the algorithm should be flexible enough to produce a variety of interesting and visually pleasing styles in addition to resembling manually produced paintings (of any form).

Another important aspect to be addressed, which was only hinted at in Section 5.1.1, is *temporal coherence*. As we intend to employ our renderer in an interactive environment, we have to deal with the aspect of animation and its inherent discrete time-steps (in a technical sense any animation, either through limitations in the recording process or limitations of the output-device has to deal with discrete time-steps). The challenge thus lies in producing discrete images at discrete times, which produce little temporal noise, i.e. they are temporally coherent.

### 5.2.2 Implementation-specific problems

A multitude of implementation-specific problems are associated with this type of renderer. The first task we have to tackle is to define some form of relation between object-geometry and brush-strokes. As with our other NPR renderers, there are no direct API calls that will allow us to set the graphics engine to paint-brush mode and to generate pixels accordingly. While it would be conceivable to modify the OpenGL pipeline at the rasterisation stage to allow for this kind of behaviour, we rather solve the problem with a more general approach. As mentioned in Section 2.1.4, Meier [51] addresses this problem by sampling the higher-order surfaces that describe her geometry and affixing brushes at sample-points. She thus replaces one representation of an object with another one, more suited for the purpose. Her approach produces images of excellent visual quality, but is computationally rather expensive, because a large number (geometry-dependent) of brushes have to be tracked and depth-sorted before the actual rendering can begin. Her justification for doing so is to avoid the so-called *shower-door-effect*, which is created, when brushes are fixed in screen-space as opposed to object-space. We therefore strive for a solution that minimises the shower-door-effect, while not being (overly) affected by the complexity or extent of the object.

Apart from the object-geometry itself, there is other object-specific information (such as colour, texture, material, etc.), which needs to be included into the final image. A common approach here is to use a reference-image, which is rendered in a realistic fashion but usually at a much lower resolution than the

final image. This idea corresponds to an artist painting her interpretation from a live scene or a photograph (even a mental image could be used for reference purposes). This means that we have to be able to generate a *reference-image* of our object, without disturbing the current scene (i.e. the background or other scene elements).

### 5.3 Solution

Our generic solution is extremely simple, yet versatile and efficient. We simply create a view-dependent reference-image based on the object-data. We furthermore define a style by appropriate configuration of Brushes, Paints and Technique and finish by painting the reference-image in the given style. This process is shown in Listing 11.

```
Render
  RefImage = ReferenceImage(Object, Viewer)
  Style = DefineStyle(Brushes, Paints, Technique)
  Draw (RefImage, Style)
End Render
```

**Listing 11 - General Painterly Algorithm**

It should be noted that apart from the generation of the reference-image, our algorithm is in essence two-dimensional (in fact, it takes some effort to ensure its proper working in a 3D context). This of course means that it can easily be used in a conventional image-processing context as a highly customisable image-filter.

### 5.4 Standard Approach (Convolution Filters)

Painterly image effects are by no means new to computer graphics and have long been used in image and video-processing applications. This means that a substantial variety of image-filters exist that will transform bitmaps into painterly-looking images. All of the commercial image-processing packages we are aware of implement these filters through convolution filters. What we mean by this is that images are interpreted as a collection of neighbouring pixels and filters work on these pixels and their relative positions, in order to achieve a given effect. In practice this means that most filters will visit each pixel of an image at least once and also take into account a window of neighbouring pixels. This window-size usually defines the Brush-size, while the traversal-algorithm and modulation-computations implicitly define *Technique* and *Paints*. Thus, if the window-size is too small, the painterly effect might be barely visible whereas if the window-size is too large, the number of pixel-calculations increases dramatically. This might not be a major factor in standard image-processing, but becomes crucial, for obvious reasons, in an interactive context. Listing 12 shows a generic convolution filtering algorithm with an unspecified filter. For edge-detection purposes this filter could simply be a convolution matrix. In order to apply an oil-style effect, this filter implements a histogramming function.

```
GenericConvolutionFunction
  Let r = half-width of convolution-window
  Let func = filter function
  Let Input_Image_Data be some two-dimensional array of pixels
  Let Output_Image_Data be some two-dimensional array of pixels

  func.Initialise(r)      // let function reset itself?
```



```

For y = 0 to Height-1    // For all pixels ..
  For x = 0 to Width-1    // .. in the image
    // generate convolution window co-ordinates
    x1 = CLAMP (x-r, 0, Width)
    x2 = CLAMP (x+r-1, 0, Width)
    y1 = CLAMP (y-r, 0, Height)
    y2 = CLAMP (y+r-1, 0, Height)
    // iterate through convolution window
    For wy = y1 to y2
      For wx = x1 to x2
        // use the current pixel in the filter-function ..
        // ..(usually some weighted average)
        func.SetData(Input_Image_Data(wx,wy))
      Next wx
    Next wy
    // apply the result of the filter-function (the function
    // might raise itself in the process)
    Output_Image_Data(x,y) = func.GetResult()
  Next x
Next y
End GenericConvolutionFunction

```

Listing 12 - Generic Convolution Algorithm

Nonetheless, we implemented a standard version of our painterly renderer with this common convolution filtering approach. The source code for the actual filter is taken from [105], and uses no MMX or Pentium style assembler-code (These would undoubtedly improve performance, but are too implementation-specific for the discussion in this thesis).

Figure 57 - a) Original Image<sup>2</sup>; b) Video filter; c) Commercial Filter

Figure 57a) shows a sample image, which we will use as our reference-image. Figure 57b) shows the above-mentioned video filter applied to the input image. The last image in Figure 57 shows the result of commercial image manipulation software with a comparable filter. Both of the latter images were generated from the original in 0.5-1.0 seconds, i.e. between 1 and 2 frames per second, with a convolution window of 10x10 pixels (i.e. equivalent to  $r=5$  in Listing 12). The three main problems we find with this (convolution) approach are:

- Low performance of a pixel-based algorithm ( $O(n^2)$ , where  $n$  dimension of image)
- Low flexibility of algorithm (most effects are hard-coded)
- Lack of natural media artefacts (i.e. native brush-strokes)

<sup>2</sup> Original Image © G. Schulz

This means that while a convolution filter can indeed produce the desired visual effect, it is in general too costly in terms of performance and the range of possible effects is too limited and/or too difficult to implement.

#### 5.4.1 Reference-image acquisition from 3D models

While Section 5.4 explained the common solution to filtering a given reference-image, we still need to discuss how we can obtain this reference-image in our 3D context.

The basic strategy is simple: We render the object from the current viewpoint and capture the frame buffer. This approach is as simple as it is flexible (we could for example render the object in a comic style and thus create a *painted comic style*), but several technical subtleties and performance issues need to be investigated:

- Minimising frame grab area
- Co-existence with backgrounds and other scene elements
- Suitable off-screen buffers

Several frame buffer grab operations exist in the OpenGL API, but they are rarely optimised and considered very costly. We therefore developed a cheap but effective way to minimise the area of the frame buffer that has to be grabbed. The method has to be *cheap* in a sense that it must speed up the grab area by minimising it, while not being too complex itself, thus negating the optimisation. Our solution therefore uses the bounding box of the object (a measure which is also used in other renderers and therefore readily available). Figure 58 illustrates how the bounding box of an object is projected into screen-space (green circles). The maxima of these co-ordinates are then established to define a bounding rectangle (red crosses) in screen-space. This bounding rectangle is then used to limit the screen-capture area. Parts of the bounding rectangle that lie outside the visible screen area are clipped automatically to screen dimensions. The `glReadPixels` command is then used to capture screen pixels. Programmer's guides and reference manuals [53] stress the fact that various commands (namely `glPixelStore`, `glPixelTransfer`, and `glPixelMap`) influence the pixel-transfer from the drawing context to client memory, so that great care should be taken to place the GL engine in a state best optimised for a given system, before the screen capture takes place. Adverse effects include type conversion, byte ordering as well as LSB/MSB order conversion, which may be performed on each pixel and thus decrease performance significantly.

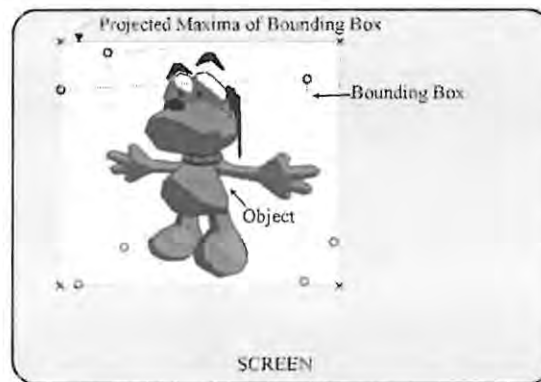


Figure 58 - Minimising the capture area

After a certain portion of the screen has been captured in the manner described above, we can use it as the input frame for our painterly style rendering. Artefacts occur if the background of the grab area (pictured white in Figure 58) is non-empty (due to a custom background or other scene-elements). In this case the painterly filter will be applied to both the object (desired) and the background (not desired). The result of such a rendering can be inspected in Figure 59. Our solution can be compared to the *blue-screening* technique commonly used in the film industry. In this technique background pixels are specially marked (i.e. have a uniform and strong colour – usually blue or green) so that non-marked pixels can easily be detected. In order to mark these pixels, we have to set a uniform background colour.



Figure 59 - Naïve implementation showing background artefacts

Since we specifically do not want to upset any scene elements that might already be present, we use an off-screen buffer for this task (this is basically any buffer, which is not visible at the time of rendering. Suitable buffers are discussed later). This off-screen buffer is first cleared with a specific masking-colour, the original object is rendered into it and then grabbed (as before), but when the painterly rendering takes place, only those areas of the captured buffer that have a colour different from the background colour are affected.

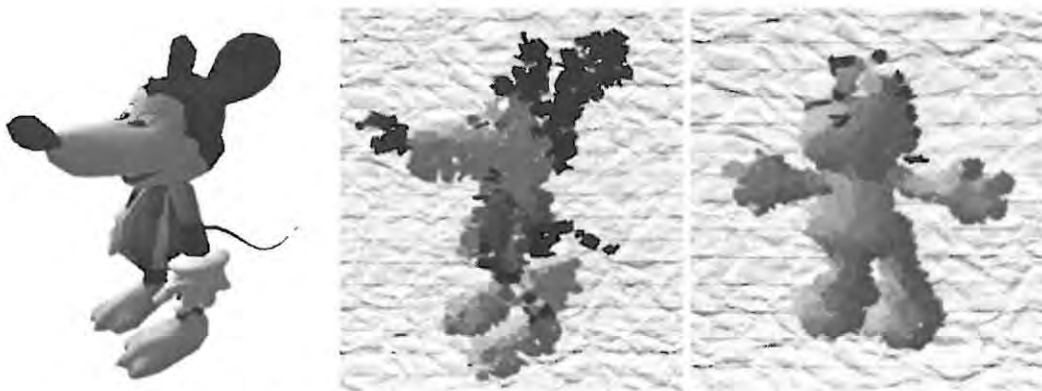
Another possible approach (which seems less complicated, as it does not need an off-screen buffer) is to simply use the stencil-buffer (this buffer discards pixels fragments depending on a bit-mask test, which the user can specify). The stencil buffer can be written to during the rendering of the original object thus creating a stencil-mask of the exact shape of the object. Later, when the painting is drawn, the stencil-test will allow only pixels within the original shape of the object to appear in the colour buffer.



**Figure 60 - Painterly Rendering using the Stencil Buffer**

A demonstration of this technique is shown in Figure 60. What should be noted are the many exact edges (on the dog's left foot, around his belly etc.). This is due to the exactness of the stencil buffer. We actually have to force some brushes with background colour to come through to break up the very straight object lines, as these lines appear unconvincing in a painterly brush-style. An advantage of this approach is that the painterly re-rendering can be applied directly over the original reference-image without the need for a previous buffer-clear operation and that the stencil-buffer can be used to distinguish between a multitude of objects. In order to use the blue-screening technique mentioned earlier, we would have to capture the stencil-buffer as well as the frame-buffer, which constitutes another expensive frame-grab operation.

Figure 61b) shows the advantage of an off-screen-buffer: If we consider for example the tail of the Mouse in Figure 61a), which is very thin we would have trouble applying a painterly effect on such a small area with the stencil-technique. The off-screen technique on the other hand enables us to re-render an object from scratch so that no reference-image creation artefacts are visible.



**Figure 61 – Painted 3D objects: a) Mouse in offscreen Buffer; b) Mouse in Screen Buffer with Paper Background; c) Dog**

For our off-screen technique, any available, non-visible buffer can be used. These include `GL_FRONT_RIGHT`, `GL_BACK_RIGHT` (we assume that the left front and back buffers are used already for a double-buffered display), `GL_AUXi` (where *i* is an integer between 0 and `GL_AUX_BUFFERS-1`).

Unfortunately the availability of stereo buffers (i.e. the set of right buffers) and auxiliary buffers is very limited. The default behaviour of the systems we tested, when non-existing buffers were used for read and write operations, is to use the last valid colour buffer. This means that the same code will run on all systems, even though the output will vary (for example the clear operation will affect the current colour buffer, so that the standard buffer will be cleared, if no additional buffers are available. Depending on the context, this issue may need to be addressed – see the use of stencil buffer above). Another possibility is the use of pixel buffers (pbuffers for short). These are buffers that actually contain their own rendering context (i.e. their own set of OpenGL state variables), that can differ in size from that of the output device context and which are located in server memory (allowing for accelerated drawing). As these are inherently non-visible surfaces, they are ideally suited for use as off-screen buffers (see [61], page 513 for details). Unfortunately, pbuffers are not yet widely available and their own set of state-variables necessitates copying of the relevant matrix-stacks in order to preserve a correct view set-up.

Each of our described approaches works and produces the desired results. The choice of the most appropriate solution depends on the given application and target system. Table 16 lists the advantages and disadvantages of the different approaches and gives pointers as to when each approach might be gainfully employed.

Method	Advantages	Disadvantages	Best used, when...
	<ul style="list-style-type: none"> <li>Needs only one Clear-Buffer operation per frame</li> <li>Easily works with multiple concurrent distinct objects</li> <li>Widely available and well implemented</li> </ul>	<ul style="list-style-type: none"> <li>Need to capture stencil-buffer for blue-screen masking</li> <li>May produce too exact edges</li> </ul>	<ul style="list-style-type: none"> <li>Many objects are to be rendered in a painterly style</li> </ul>
	<ul style="list-style-type: none"> <li>Easy to implement</li> <li>Same state-variables throughout</li> </ul>	<ul style="list-style-type: none"> <li>Clear-Buffer operations on large buffers</li> <li>Very limited availability</li> </ul>	<ul style="list-style-type: none"> <li>State-variables change often</li> </ul>
	<ul style="list-style-type: none"> <li>Custom size (esp. smaller than visible Buffer)</li> <li>Never visible (i.e. Clear can be performed when convenient)</li> </ul>	<ul style="list-style-type: none"> <li>Need to import state variables (Matrix stacks &amp; Lights) from Main rendering context</li> <li>Fairly limited availability</li> </ul>	<ul style="list-style-type: none"> <li>Few State-changes occur</li> <li>Frame-grab operations are very expensive</li> </ul>

Table 16 - Pros and Cons of Background preservation Techniques



## 5.5 Optimisation (Textured Brushes)

We feel that part of the reason why the convolution painting approach is so expensive and inflexible is because it presents a fairly abstract solution to a very tangible problem. If it were more intuitive how the design of the filter-function influences the resulting image, it would be far easier to construct the right filter for a given purpose.

In order to overcome these limitations, we decided to approach the problem on a more basic level. By moving away from a programmer's perspective and into an artist's viewpoint, we immediately came up with a totally different approach. We remember the concepts of *Brushes*, *Paints* and *Technique* that were somewhat merged and hard to distinguish in the convolution approach, and seek a more direct implementation.

### 5.5.1 The Brush

We start by modelling the Brush. As with the sketch renderer, we use textured quads and blending to produce the desired output. We therefore define a brush object with the following properties:

- Position (unity dimensions)
- Rotation (in canvas plane)
- Size
- Shape (texture)

For our purpose, we find it useful to extend the standard idea of brush-shape, which we depicted top-most in Figure 62 and which is most closely related to the cross-Section of the bristles of a brush. The next image in Figure 62 is obtained by moving the brush-shape over some surface. We call this a *brush-stroke*. To simulate colour draining from the brush onto the surface, we define a transparency mask, with increasing transparency in the direction of the stroke. Finally, if we combine the transparency-mask with the brush-stroke, we get the bottom-most image in Figure 62, which is our complete brush-stroke. This is what we define as our brush-shape and which we use to render our images. That means that each brush contains in itself not only the shape of the original brush, but also the stroke with which it is applied to the surface as well as its capacity to hold and emit paint. Figure 68 convincingly demonstrates how our brush-shape extension creates the illusion of applied brush-strokes.

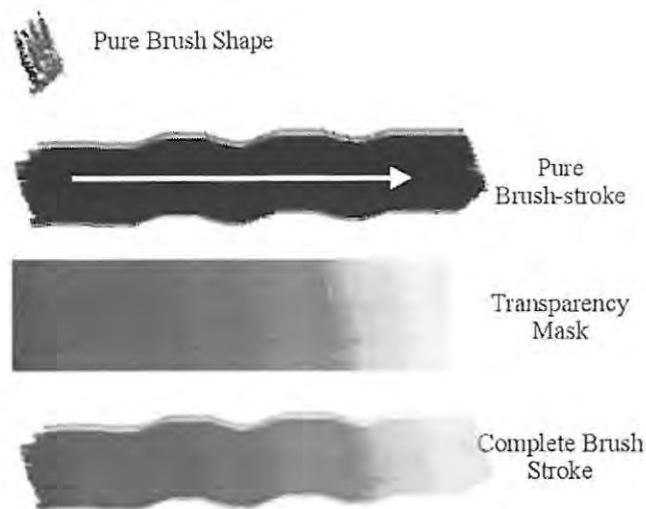


Figure 62 - From Brush Shape to Brush Stroke

By randomly distributing the brush positions inside a unit square, the brush position can be multiplied by the dimensions of the input image and the correct input colour sampled. We then multiply the brush position by the dimensions of the output image and render it onto the output image (after all necessary transformations have been applied). This yields a convincing painted look, but the static nature of the brushes is visually uninteresting if not disturbing (i.e. this results in the above-mentioned *shower-door effect*). To remedy the situation, we apply a pseudo-random Brownian motion to the brushes and therefore extended their properties by the measures of speed and acceleration. This approach is depicted in Figure 63.

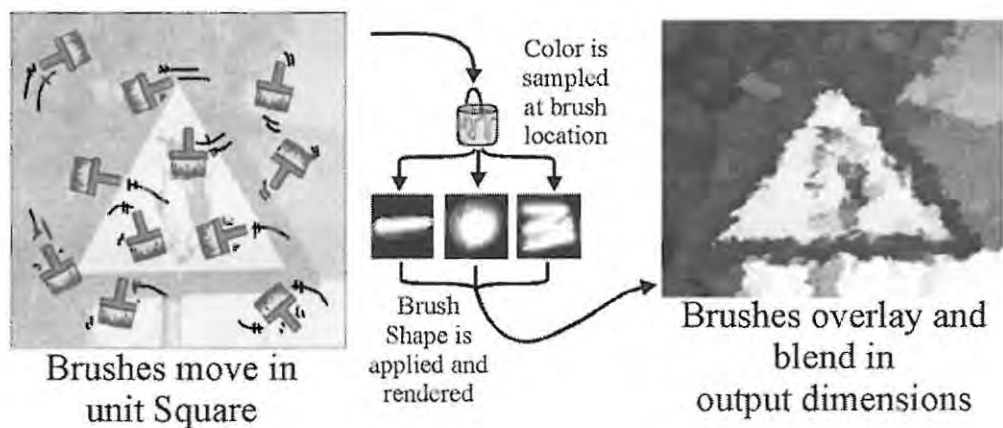


Figure 63 - Oil Painting using Textured Brushes

A multitude of associated issues needs addressing:

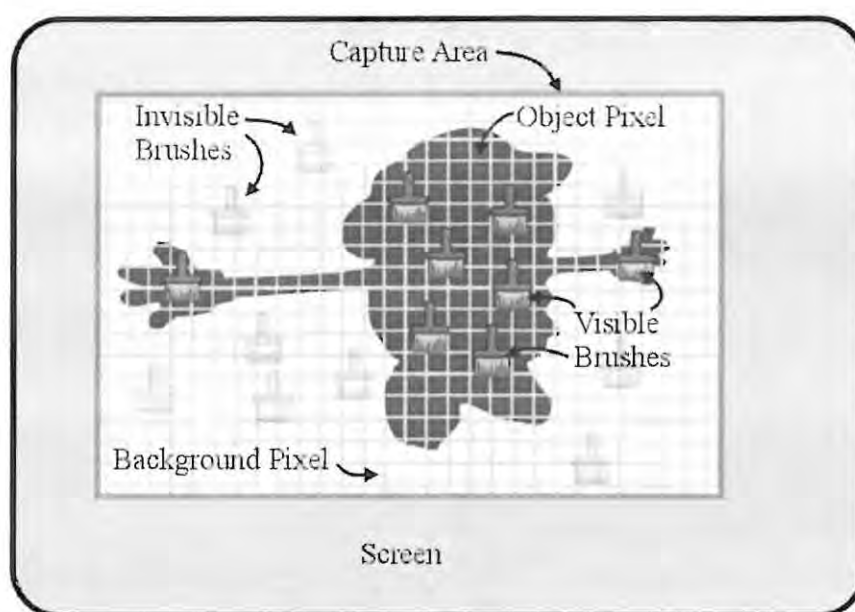
- Number and size of brushes
- Edge difficulties (suicidal brushes) and invisible Brushes

In general, the number and size of brushes should be chosen so that the statistic coverage of the output image is as high as possible (the visual quality of the output image is greatly compromised if background-pixels show through, so that the fill area should be maximised). Let us consider Figure 64. If we suppose that the percentage of the captured frame area occupied by the object is for example 25%, then we can assume that 25% of randomly distributed brushes are situated over object-pixels (as opposed to background-pixels). If we use our blue-screen masking technique described above, it follows that only 25% of our brushes will be rendered (i.e. visible), even though we have to animate and distinguish all of them. We must therefore keep the number of brushes over object-pixels as high as possible. We achieve this by monitoring brushes and checking if they move out of a object-pixels region onto a background region. Once this condition is detected, the brush's movement-direction is inverted until it finds itself back inside a visible region or until a time-out period ("*time to live*") has elapsed. The latter is used to make sure that brushes do not spend too long finding their way back into visible regions. After a brush has spent a given number of frames invisible, it is respawned (i.e. re-initialised) at a random location. Section 5.5.1.4 explains why the variable, which counts the time a brush is lost, should be initialised to a random value less than the maximum time to live.

By determining the visibility of brushes, we easily obtain an accurate frame-to-frame count of how many brushes are visible. By sampling the captured-frame area at regular intervals, we can establish an estimate of the pixel-area covered by the object. These values together with the screen-size of the captured area allow us to determine the necessary brush-size to guarantee a desired statistical brush-coverage in screen-space. To recapture: Let  $P_B$  be the percentage of visible brushes and  $P_A$  the percentage of pixels occupied by the object in the frame-capture area. Furthermore, let  $B$  be the number of brushes and  $A$  the area of the frame-capture region, then

$$A_B = \frac{P_A \cdot A}{P_B \cdot B}$$

is the screen-area each brush should occupy in order to completely fill the object-area with brushes. Because brushes are distributed randomly this value should be scaled up slightly to increase the probability that brushes overlap, thus creating a more seamless picture.



**Figure 64 - Definitions: Visible Brushes, Object-Pixels, Background-Pixels, and Capture Area**

Another approach to maximise the fill area is to choose a different brush animation function. If this function can guarantee that brushes do not overlap (for example if brushes moved around circles placed on an evenly spaced grid), the statistical fill-area increases and fewer brushes can be used. The advantages of this approach have to be weighed against the adverse effects of easily detectable pattern motion and the overhead in implementing it.

As brushes move inside the unit square they will reach the unit boundaries and try to cross them (unless a motion function is defined which implicitly forbids this). This situation can be resolved in a number of ways:

- The brush dies and is respawned at a random location inside the unit square
- The brush bounces off the edge
- The brush wraps around to the opposite side retaining its motion

As brushes have a finite speed, the second solution is problematic. Computing a proper reflection while taking discrete time-steps into account is fairly costly for a large number of brushes. A naïve approach using offsets and not considering timing issues is likely to trap brushes along edges. To wrap brush positions around the unit square may in some cases also result in problems, as the speed (and therefore travelling direction) of our brushes changes dynamically. This can result in brushes flickering between opposite edges, especially if the brush travels parallel and very close to an edge. We work this problem into the same solution we discussed above and treat brushes that attempt to move over the unit-edges as invisible. They are naïvely bounced off the edge and given a certain number of frame to re-appear in a visible region, before being respawned, thus implementing a combination of the first two suggested solutions.

If a large number of invisible brushes fail to reach visible regions within their given time to live (say  $x$  frames), then they will all be respawned after that given period, resulting in the output flashing every  $x$  frames (this effect will be very pronounced during an initial stabilisation phase and then fade into statistical noise). To avoid this flashing, we must initialise the counter, which keeps track of time spent invisible, to random values. This will still see many brushes respawning in the initialisation phase, but it will happen desynchronised (i.e. in different frames and not simultaneously) (see Section 5.5.1.4 for results). Listing 13 shows the general algorithm used in our painterly renderer with textured brushes.

```

Update (Brush B)
  B.Position = MotionGenerator.UpdatePosition(B) // determined visibility
  If Not Visible(B) // if not visible ...
    If B.Visible = True // ... but used to be visible
      B.Motion.Invert() // Invert movement direction
      B.Visible = False // we're as invisible
      B.TimeToLive = MaxTimeToLive // Start count-down
    Else
      If B.TimeToLive==0 // lost for too long
        B.Respawn() // respawn brush
      Else
        B.TimeToLive-- // another frame not contributed to
      End If
    End If
    Return // Brush is not visible
  End If
  Brush.Visible = True // now count as visible
  B.Style = StyleManager.UpdateStyle(B) // can influence, colour, shape, ...
End Update

Draw (RefImage, Style)
  Let P = Projected BoundingBox
  Let B = array of Brushes
  Let Reference_Image = Pixel-array to hold reference image

  TransformToPosition(BlitBackPosition(P)) // set general position offset
  For all Brushes in B
    Update(B[i])
    If B[i].Visible
      Draw B[i] // TextureMap(), Zipping(-), Depth-Buffer(-),
    End If // Blending(+), Culling(-)
  Next Brush
End Draw

```

**Listing 13 - Painterly Rendering using Textured Brushes**

Animation G shows all of these effects in action (Brownian motion of brushes, home-seeking technique of brushes, overall movement of brushes corresponding to object's screen-position, colour-averaging of brushes, and self-adjustment of brush size)

#### 5.5.1.1 Qualitative Results for Number of Brushes

The single most influential factor of this renderer in terms of affecting both quality and performance is the number of brushes that are rendered to replicate the reference-image. Seeing as we want to guarantee that the total object is covered with brushes, we have to increase the brush-size with decreasing number of brushes. This results in loss of visual detail compared to the reference image. Conversely, increasing the number of brushes allows us to decrease the size of each individual brush and finer object detail emerges. This relationship is visualised in Table 17. Personally, we believe that objects re-painted with 2000-4000 brushes look most convincingly painted. If the brush count is considerably below that, too much object



detail is lost and repetition of similar brushes may in cases be noticed (we usually only use three different brushes, which is under normal circumstances well hidden by the blending operations). If the brush count moves well above 4000, the painterly effect of the brushes is increasingly lost. In general, a user of our system will have to decide on the desired resolution and adjust the number of brushes accordingly. It should be noted, that apart from the number of brushes, no other variables were changed, proving that our automatic brush-size adjustment (as a function of visible brushes, screen-capture area and object-pixels in that area) works seamlessly and produces visually coherent results.

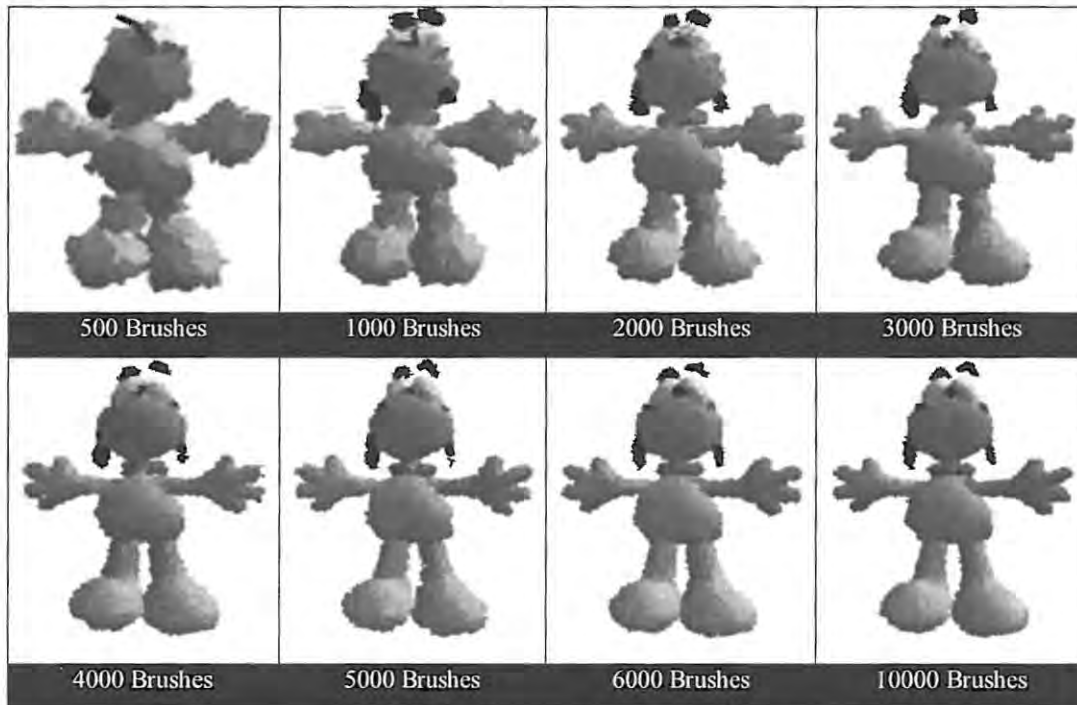


Table 17 - Object Resolution vs. Number of Brushes

#### 5.5.1.2 Quantitative Results for Number of Brushes

As already mentioned earlier, the performance of our painterly renderer is only very slightly affected by the complexity of the object used to generate the reference-image. Other factors play a much more influential role. As we see in Figure 65, the individual traces are fairly straight and almost parallel to the x-axis, indicating their independence of the number of triangles. It should be noted that the increase in number of brushes is constant except for the first trace and the last two traces, the latter of which show extreme values. The 1-Brush trace behaves slightly erratically, which we reason is due to the random position of the one brush. As the brush is scaled to cover most of the object, its position determines how much of it is visible and how often it becomes invisible. Nonetheless this trace defines a fairly rigid upper limit, the value of which is primarily affected by the cost of the screen-capture operation. Comparing the performance of the painterly renderer with one brush to the standard renderer we find that in average the screen-capture operation alone causes a roughly 7-fold slow-down.

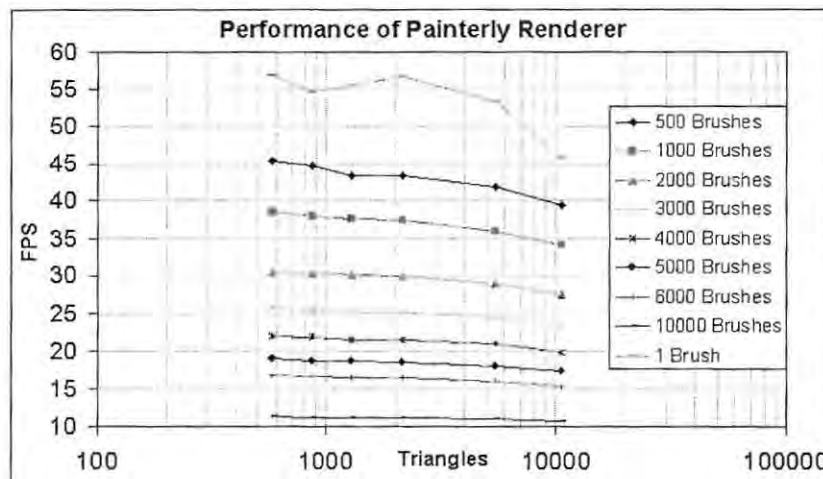


Figure 65 - Performance vs. Number of Brushes

Considering this, the effect of rendering more brushes on performance is actually not very large. As we stated above, our personal opinion is that reference-images re-painted with 2-3000 brushes look visually most convincing. Figure 65 indicates that this allows for a comfortable refresh-rate of 25-30 frames per second. Even for 10000 brushes the performance does not drop below 10 frames per second proving the real-time performance claim of our painterly renderer.

#### 5.5.1.3 Quantitative Results for Object Speed

In order to maximise the number of brushes that are hovering over object-pixels and are therefore visible (as opposed to those that hover over background pixels and are invisible), we supply each brush with a tiny amount of intelligence – just enough to find their way home (i.e. over object-pixels). If a brush is lost, i.e. marked invisible, due to moving into background-pixel regions, it will try to retrace its way in the hope that it will rediscover *visible* space. Failure to do so after a pre-determined amount of time will result in instant death, compensated by a similarly instantaneous reincarnation. It follows that for stationary object-pixels, the home-finding task is relatively easy, whereas a moving target is much harder to find. If the brushes were randomly distributed (as was the case before our optimisation) the chance of any given brush being visible would be equal to the ratio of object-pixels to total screen-capture pixels (i.e. the *object fill-ratio*) and independent of the object-position inside the capture-area. By applying this optimisation, we create local regions of high brush-densities, which are dependent on the object-position. As the object moves inside the capture-area, these high-density regions have to follow accordingly.

To illustrate this fact, we have performed a series of tests, in which an object (the dog) is rotated at various constant velocities and measured the effects on the number of *visible brushes*, the number of *respawned brushes* and the *object fill-ratio*. Table 18 shows the results of these tests. The top-most trace in each graph represents the number of visible brushes. The middle trace represents the object fill-ratio and the bottom-most trace represents the number of brushes that have to be respawned. The horizontal axis measures units of time and is scaled to show a full rotation (i.e. the slower rotations will have many more samples).

The first graph shows the results for a stationary object. Several interesting facts become apparent. Firstly, the object-fill ratio is constant as we would expect, because neither the dimensions of the screen-capture area change, nor do the number of pixels representing the object. The object (from the starting orientation of the rotation) fills 22% of the capture area. The visible brushes step up towards, but never reach 100%. This means that brushes will find their way into the visible region and then try to stay there. In accordance with theory, at the beginning, when brushes are totally randomly distributed, only about 22% of them are visible. The spikes in the respawn-trace show that every 10 frames an exponentially decreasing number of brushes have to be respawned. 10 frames is the *time to live* for a lost brush in these tests. These spikes appear as flashes in the rendered image and are visually disturbing. This issue will be addressed in Section 5.5.1.4. We can note that with each respawning-spike, the number of visible brushes surges upwards accordingly, indicating that at least a certain number of brushes have by chance relocated to a visible area. As time progresses and almost all brushes are visible, almost no brushes have to be respawned.

In the following graphs, the test-object is rotated at a constant, finite speed, indicated below the graphs (in radians per frame). We can see that the object fill-rate trace is the same for all graphs. This is not surprising, as this measure is totally independent of the brushes, and rather a function of a given object's geometry and the current viewpoint. The top-most trace, indicating visible brushes, has the same basic shape throughout, but with more and more pronounced dips as the speed increases. The general shape of the curve is a function of the geometry of the object and its rotational axis, while the amplitude of the dips is related to the speed of the brushes (i.e. how fast they can find their way back) and the speed of the object (i.e. how fast it can lose them). We identify a similar behaviour for the respawned brushes. The basic shape of this trace is the same throughout, but with amplitudes rising slightly as the object's rotational speed increases.

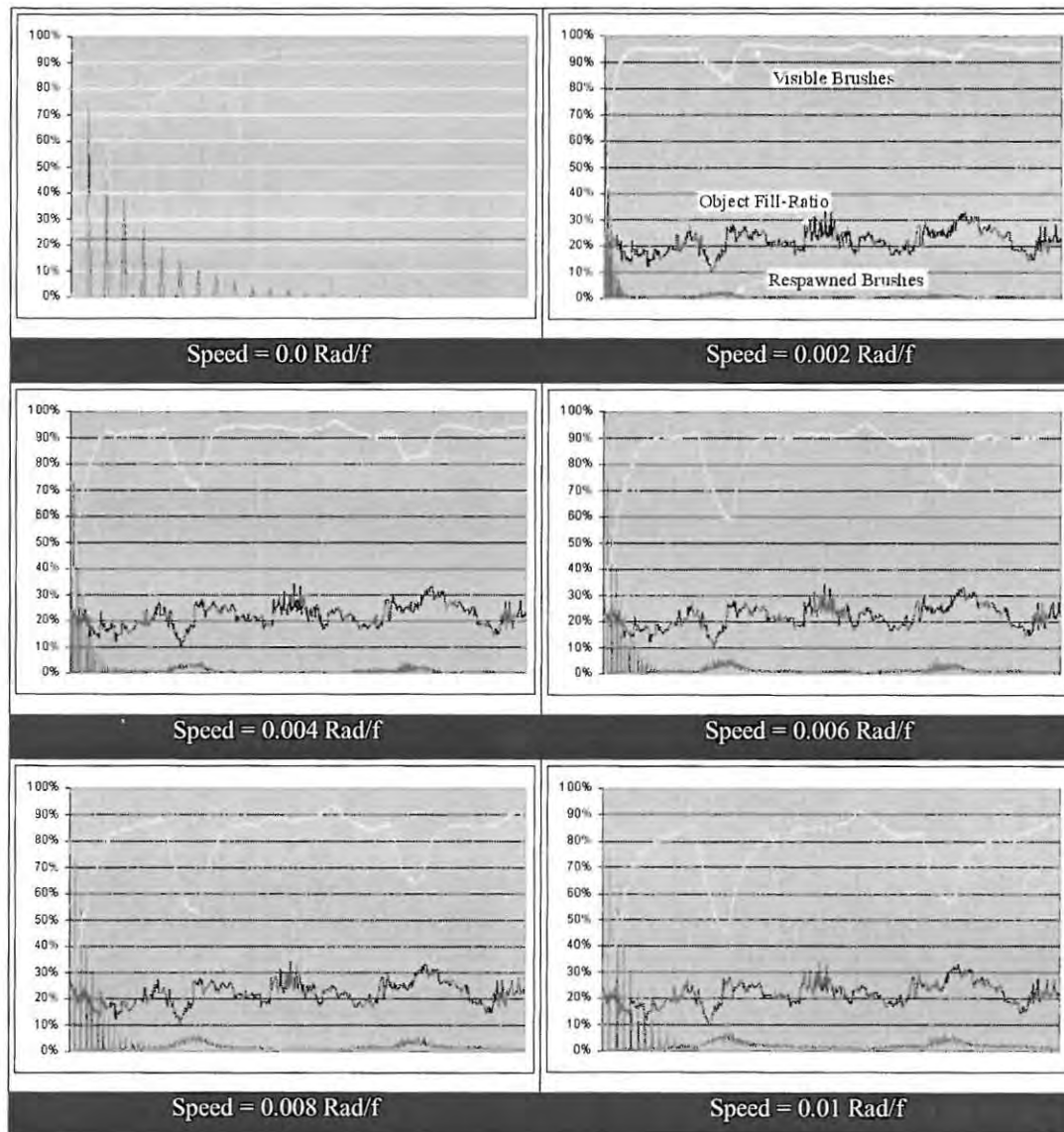


Table 18 - Effects of Object Speed on Painting Behaviour

To show that our optimisation also works for extremely high velocities and to demonstrate long-term behaviour, we performed another test with angular speed of 0.5236 Radians/frame (=30 Degrees/frame), ensuring that any periodic behaviour resulting from the rotation lies beyond our arbitrary 10 frame *time to live* mark (i.e.  $360/30 = 12$  frames per full rotation). Object symmetry may also create periodic behaviour, but brushes have no information about this and can therefore not knowingly take advantage of this fact. The traces for this test are shown in Figure 66. On the x-axis in this Figure we show not just one, but 100 rotations. The object fill-ratio is, as expected, fully periodic and in fact just an extremely compressed version of the equivalent traces in Table 18, repeated 100 times. Two very interesting facts emerge. Firstly, the number of visible brushes still rises quickly and settles around  $70\% \pm 10\%$ . This means that despite the high angular velocity of the object and its constantly changing projection, brushes find their way into regions of the capture-area that are visible most of the time. The improvement in visible brushes of our method is still about 50%. Secondly, the respawn-trace sinks below the 1% mark after only 5

rotations. This means that brushes either move into regions, which are visible most of the time, or into regions that are frequented often enough by object-pixels. In the latter case, high object speed will even prove beneficial.

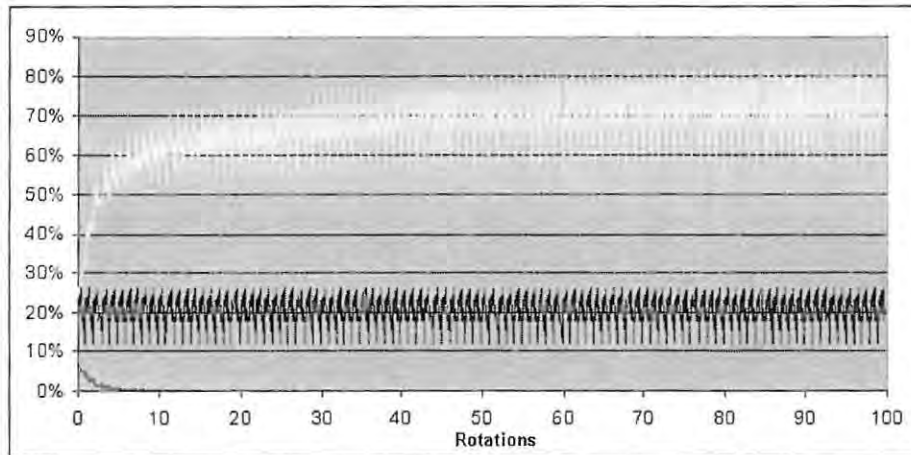


Figure 66 - Extremely High Speed

To measure the success of our optimisation, we would like to define the desirable results. We want to keep the number of visible brushes as high as possible. The greater the difference between the visible-brushes trace and the object fill-rate trace, the greater the gain of our optimisation. Respawnng brushes is not very expensive, but can create visual artefacts. For this reason this count should be kept as low as possible. Our tests prove that after a short initialisation period, our system becomes relatively stable and a considerable gain in the number of visible brushes can be achieved even for very fast-moving objects. It should also be noted that the worst case is a (non-practical) totally random distribution of object-pixels in the capture area in each consecutive frame. In this case our system would exhibit the same characteristics as a non-optimised system. As our optimisation is easily and efficiently implemented, its potential gain far outweighs its incurred overhead in this rather contrived situation.

#### 5.5.1.4 Quantitative Results for Respawnng Behaviour

As mentioned in Section 5.5.1.3, the respawning-spikes, resulting from synchronous re-initialisation of a large number of brushes, is visually noticeable and distracting. The very reason that this effect is so noticeable is due to the large amplitude of the spikes. This in turn derives from the fact that re-initialisation occurs synchronously every  $x$  frames, because all brushes are initialised with the same *time to live* (e.g. 10 frames). Our solution is therefore simply to initialise brushes to random *times to live* upon first creation. This desynchronises respawning, even though still the same number of brushes is respawned over the same period of time (i.e. the integral of the curves is the same). The effect is that the traces for both the visible and the respawned brushes exhibit a much smoother shape, equating to visually much smoother animations.



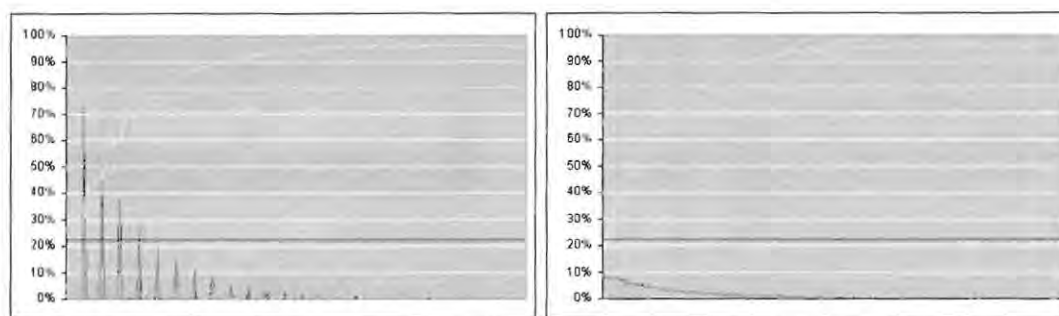


Figure 67 - Respawn Behaviour: a) Synchronised; b) Randomly desynchronised

### 5.5.2 The Paint

The idea of a palette can easily be realised by quantisation of the colours of an input image (see rule a in Figure 69). This can be achieved with an appropriate function or via a lookup-table to create effects such as *sepia-colouring*, *false colours* and *neon colours* to name but a few. A side-effect of this is usually greater contrast in the output image so that individual strokes become more distinguishable. The opposite effect can be achieved by averaging colour values over time (we do this by extending the brush properties with a colour attribute and averaging over a given number of frames). We find that while this produces a uniform and smooth appearance, fast moving brushes can carry unwanted colour into regions, where this colour is usually not found (see the brush-fading effect in Animation G). In cases this might be used to indicate manual flaws, in other cases it might be undesirable and the number of frames over which averaging occurs should be decreased.

Transparency and thickness of a type of paint can be simulated by changing the transparency of the texture maps used to draw the brushes. Higher transparencies can be used for watery colours, while lower transparencies can simulate thicker colours like oil. The different bleeding and mixing characteristics can be approximated with different blending functions, and again transparency.

### 5.5.3 The Technique

The style produced by our method is mostly influenced by the following factors:

- Brush shape
- Palette
- Orientation of brushes
- Local choice of brushes

Different strokes imitate different styles. This is why an image rendered with mostly long thin strokes will look very different from one that uses thick round blotches. A variety of brushes can be used to produce a rather homogenous appearance, while using just one brush may be used to imitate a style like Pointillism. The orientation of brushes is another important factor. Some artists might apply strokes in only one direction (like in the cross hatching shading in Section 4.6.2), along object contours or arbitrarily. For

more sophisticated methods like the contour-tracing, filters (like edge-detection or convolution filters) will have to be applied to the input image in order to orientate strokes accordingly. This is actually not as costly an overhead as one might think. OpenGL for example provides numerous image manipulation functions in the form of Convolution Extensions (e.g. `glConvolutionFilter2D()`, `glCopyConvolutionFilter2D()`, and `glSeparableFilter2D()`; see [50], node 196 for details) which work similar to the `glDrawPixels()` command, except that the convolution parameters are applied, before rasterisation. Where these are not available, they can be emulated with the accumulation buffer ([50], node 195). Using these convolution filters a variety of filters can be implemented (see [50], node 167 for an overview; node 198 for line detection filters and node 203 for gradient detection filters. Several others are given as well), several of which we consider useful in applying painterly styles to our brush strokes. If geometrical information about the content of the input image is available (as is the case in Section 5.4.1) it can be used (instead or in addition to the output of convolution filters) to align brush strokes along object geometry, take light direction into account and the like. Another possible variation is the orientation of brush strokes according to brush location (but independent of the input image). For example a sine function could be used with the x-coordinate as the argument, thus creating brush waves on the canvas.



**Figure 68 – Landscape with various Brush Orientations<sup>3</sup>: a) Random; b) Angled; c) Circular**

Figure 68 shows several brush orientation functions: a) orientation changes as an arbitrary function of position, appearing random; b) orientation is constant (35 degrees); c) orientation is  $\text{atan}(y/x)$ .

Similar to the localised choice for the orientation of a brush, the texture used for a given brush may change according to location. This means that the outline of an object could be painted with different strokes (textures) from other regions. The background could yet again use other brushes. As above, brush shapes can also be chosen as a function of brush position or even local colour of the input image. We demonstrate some of these ideas in Figure 69. Several rules have been applied to the image and three different brushes used.

<sup>3</sup> Original Image © G. Schulz

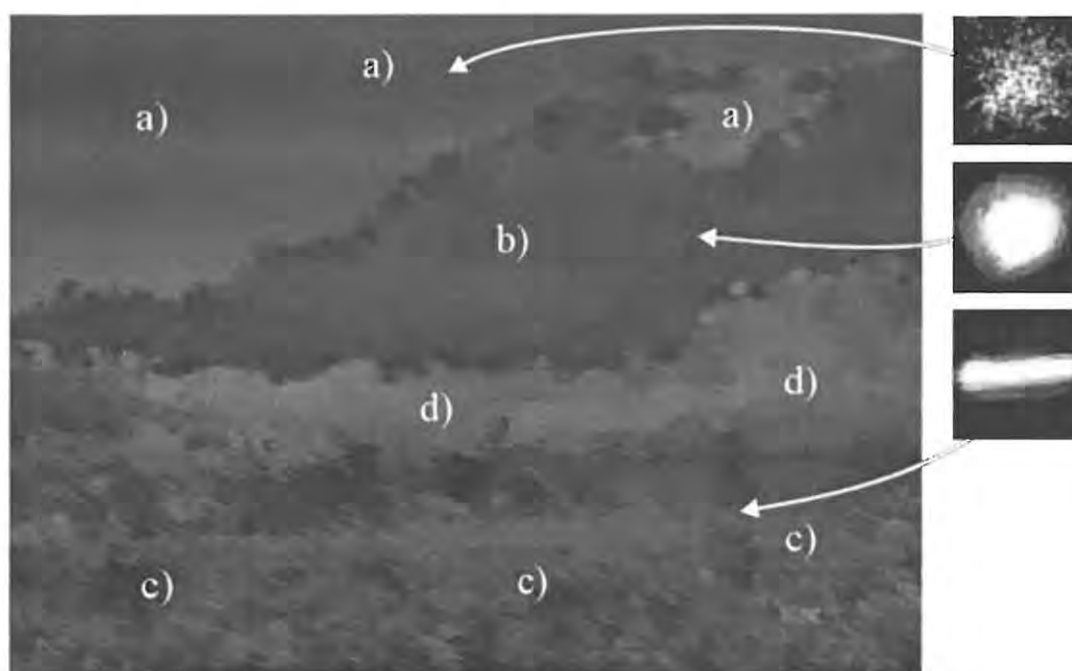
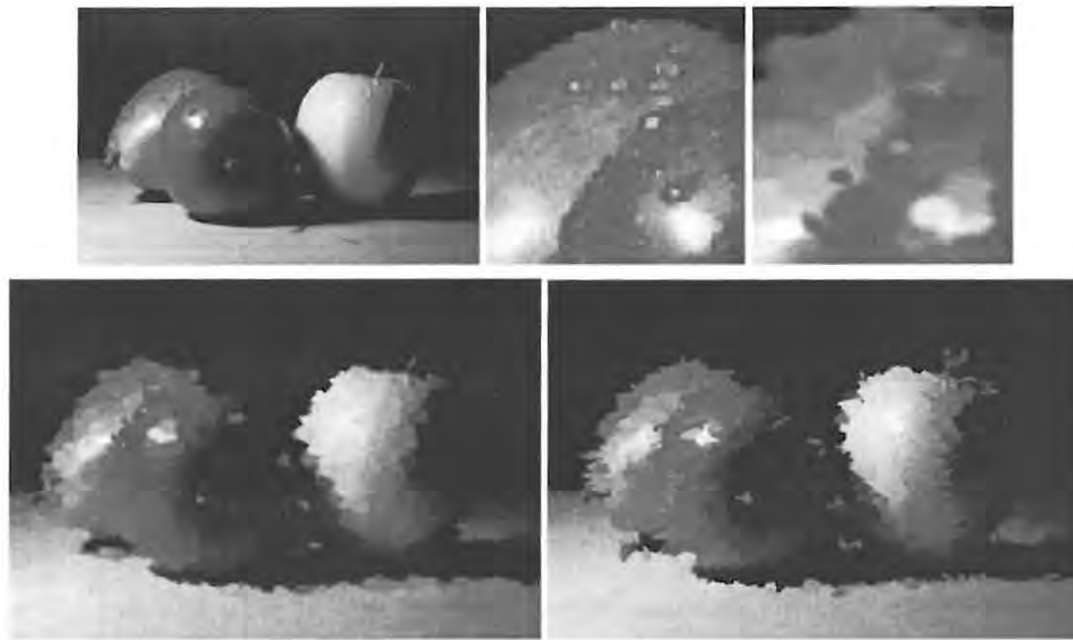


Figure 69 - Demonstration of multiple effects

In the regions labelled a) (identified by being mainly blue), we quantise the brush's colour to simulate a palette that contains only 10 shades of blue, resulting in some colour-bands in the sky areas. Additionally, these regions are painted with the *Spatter* brush. Regions c) are mainly green. Here we apply the *Stroke* brush and set the drawing angle at a constant 30 degrees. In addition, all blue and red colour-content is eliminated, enhancing the saturation in that part of the image. In regions d), neither blue nor green colours prevail and we apply the *Blotch* brush after converting the brush's assigned colour to greyscale, except if its brightness is above a certain threshold, as in Region b), in which case we apply a shade of red (i.e. what used to be clouds). While all of these rules are totally arbitrary and of no particular artistic value, they are very easily implemented and show the great creative potential, that can be realised with our method. It should be noted that, while our rules were hard-coded in a few lines of code, they could easily be made part of a scripting language, thus eliminating the need for re-compilation of source-code and facilitating the exchange of styles developed in this manner.



**Figure 70 - Brush Variations:** a) Original<sup>4</sup>; b) Detail of Spotlight; c) Same Detail with Dot-Brush; d) Dot-Brush; e) Stroke-Brush

Figure 70 shows the reference-image (a), a painterly re-rendering using our *dot* brush (d) and our *stroke* brush (e). A detail of the highlights on both apple and pear (b) rendered with the dot brush is shown in (c). Apart from demonstrating the creative possibilities of our renderer, we also would like to draw attention to the dimensions of the in- and output images (the images in Figure 70 are scaled for layout purposes).

Image	Width	Height	Pixels
Original	383	249	95,367
Dot	620	519	321,780
Stroke	528	376	198,528

By using blending, bi-linear interpolation (texture smoothing) and variable brush sizes, we can interactively change the size of the output image without affecting its quality. We can also work with a constant brush size and increase the number of brushes to compensate increasing output dimensions, again obtaining a perfectly rendered image every time. For most practical purposes the inherent loss of fine image detail allows us to use reference images of fairly low resolution.

## 5.6 Extensions

### 5.6.1 Real-time Video Oil-painting

Since we implemented our painterly renderer as a 2D filter approach, the reference image can come from any type of 2D image sources – a fact of which we made use in the previous Section, by using the colour buffer of a 3D renderer as the reference image. Static images have been used in Section 5.4, helping to describe our technique. Another popular application area is that of video special-effects (as can be found

<sup>4</sup> Image Source: Internet

in many popular video editing suites or in research, e.g. [32]). While commercial video software usually applies oil-filters in off-line batch processes, Hertzmann and Perlin claim *real-time* results of about 4 seconds per frame. Our method is limited by two factors: the refresh rate of our video capturing (using standard Windows API calls, this is usually 15 fps, mostly independent of the hardware) and resizing of the input image to comply with OpenGL constraints if the input image is to initialise the output image (both width and height have to be some integral power of two). Actually the byte ordering of our particular capture driver also has to be changed, but this can be done as part of the image resampling. Figure 71 shows our demo application re-rendering a popular South African children's television programme at about 12 frames per second (the indicated 31 fps is target rate and can only be achieved in overlay mode – i.e. by allowing the capture card to draw directly into the screen memory), thus operating at almost full capacity of the capture driver. As we discussed above the dimensions of the output rendering can be decoupled from those of the input image and we could render the scene in the full screen if we wanted to without much performance loss (the fill-rate of the accelerator card is of greatest influence here).

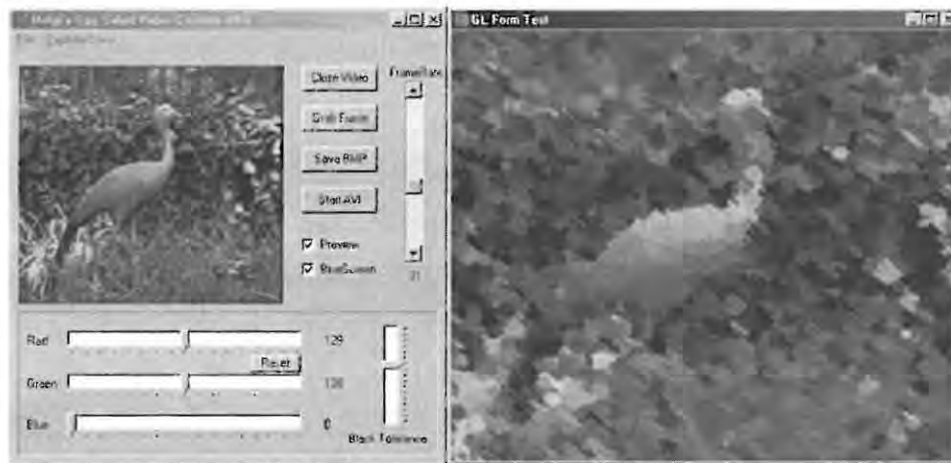


Figure 71 - Video Oil Rendering

Some optimisation can be effected when performing live video oil rendering: The sample image resolution can be adjusted to the minimum necessary. By taking advantage of inter-frame coherence (a fact that is exploited by many video compression algorithms), we can spare the background clear, simply render over the old scene and cut down on the number of brushes rendered. This provides a full visual without background artefacts. A lower brush count will update only part of a given scene, but motion detection can be performed to detect changing scene elements and concentrate the updating efforts in those areas. The content of the video sequence is also of importance: Fine detail like writing will most likely not be readable. Fast moving content will also appear blurred and, depending on the number of brushes used, produce confusing results. In general though, our method can produce convincingly painted scenes at interactive rates.



## 5.7 Results

### 5.7.1 Qualitative Comparison of Approaches

In Table 19, we show some examples of objects rendered with the default renderer (left column) and their painterly equivalents (middle column = convolution filtering, right column = textured brushes). While it is not for us to judge on the aesthetics of the results, we can note the following. The convolution filter implementation has difficulty reproducing thin object detail. The arms of the cowboy in the second row for example are totally cut-off.










Default	Convolution Filter	Textured Brushes
		
		
		

Table 19 - Visual Comparison of Painterly Approaches

Each column is rendered with a constant set of rendering variables to allow for better comparison between objects. Nonetheless, we can easily change the appearance of the textured brushes implementation totally by only changing the brush shapes it uses. To achieve a similar effect with the convolution filter implementation, we would have to re-design and re-compile its source-code.

### 5.7.2 Quantitative Comparison of Approaches

As we noted above, the performance of the convolution filter approach is so low, that it cannot successfully be used in an interactive context. For this reason, we refrain from a full comparison, and go into more detail on the performance of our textured brushes approach in Section 5.5.1.2. The main reason we included the convolution filter approach in our investigation is its common usage in commercial applications and the fact that alternatives, apart from ours and similar approaches, are few and far between. It should suffice at this stage to state that the convolution filter approach performs at  $0.363 \pm 0.021$  frames per second in our test set-up.

## 5.8 Summary

In this Chapter, we have discussed the relevant issues pertaining to painterly rendering at interactive frame-rates. Firstly, we found that it is most helpful to describe how paintings are created instead of how they look. By doing so, we identified three main ingredients:

- Brushes
- Paints / Colours
- Technique

We used the notion of a reference-image to dissect our painterly algorithm into several logical steps (acquisition of reference-image, definition of style, re-rendering of reference-image in given style), which allow it to be flexibly implemented and facilitate a filter-concept design. This means that a variety of input-sources can be processed into a painterly style without modifications to the main algorithm.

We demonstrate how image-processing filters are commonly implemented using a convolution approach and commented on its disadvantages, namely:

- High computational complexity = low performance
- Hard-coded filter-design = low flexibility
- Lack of natural media artefacts = mechanic look

Still, we were able to show how our painterly ingredients pertain as well to most convolution solutions.

We address the general problem of obtaining a reference-image from a 3D model. This is potentially an expensive task due to some generally un-optimised operations (frame-grabbing). In order to maximise the performance of this stage, we minimise the frame-grab area by means of a projected bounding-box approximation. Related to this topic, we introduce the various on and off-screen buffers that can be used to implement the reference-image acquisition, discuss their advantages and disadvantages and give general rules about the application-context that each buffer is best used in.

Next, we introduce our version of a textured brushes solution to the problem, which implements the painterly ingredients in a more intuitive fashion. Brushes are modelled as textures, making their appearance and contribution to the rendering process more accessible. Furthermore, we show how the simple notion of a brush-shape can be extended to implement artistic strokes and supply each brush with attributes such as:

- Position
- Direction
- Shape
- Size
- Paint-emission function
- Opacity

These are easily defined and adjusted to suit the user's need in creating stylistic variations. Since static screen-space brushes are claimed to produce an undesirable effect, called the *shower-door-effect*, and brushes fixed in object-space are associated with the requirement of a large number of brushes as well as a considerable depth-sorting overhead, we investigate a new hybrid-approach. Our brushes are neither fixed in object nor screen-space, nor do they appear at random positions. Rather, they move smoothly in a semi-random Brownian motion, guided by but not fixed to, object-pixels in the capture-area. We devised various movement-rules that allow brushes to find their way back over capture-regions occupied by object-pixels (the ratio of which is the *object fill-ratio*), which we call *visible regions*. Brushes that stay invisible for longer than their pre-determined time-to-live, expire and are respawned (re-initialised). Considering that the percentage of visible brushes for a completely random distribution of brushes is equal to the object fill-ratio, we easily obtain up to 99% visible brushes with our approach. This means that we can use far fewer brushes to achieve the same fill-effect. We even showed that for fast moving objects, for which our approach is less effective, a considerable gain in the number of visible brushes can be achieved.

Paints and colours are discussed in the form of palettes, which can be used to change the colour-mapping of the reference-image. This can be implemented using functions or look-up tables and its integration into our implementation is straightforward.

In terms of the variables that affect the stylistic nature of our rendered images (i.e. what we termed previously as *Technique*), we consider the following:

- Brush shape
- Palette
- Orientation of brushes
- Local choice of brushes

and show examples of their effect on the produced output, thus demonstrating the large potential of possible styles that can be achieved with our approach.

As an extension, we reveal how, owing to its filter-design, our implementation can easily be plugged into a video-capture application to produce painterly images from a live-video feed in real-time. We also present optimisations that can be applied in this context by exploiting temporal coherence.

Next, we verify the validity of our approach by listing the qualitative and quantitative results from our standard test set-up running the painterly renderer. Amongst other results, we prove that our brush-movement rules are highly successful in obtaining the highest visible brush-count possible. In our tests we achieved between 70-99% brush-visibility for slow to extremely fast-moving objects. We identified the major cost-factor in our approach, namely the screen-capture operation and suggested that about 2-3000 brushes are necessary to convey a realistic painterly looking visual. The performance in this region of brushes lies around 25-30 frames per second and even for a large number of brushes (10000) does not drop below 10 frames per second.

In conclusion, we have shown that painterly rendering of 3D objects can be achieved at interactive frame-rates and with a convincing painterly look. Furthermore we have identified some key-aspects of painterly rendering and managed to translate these into tangible and adjustable variables. We introduced several optimisations like screen-capture minimisation, and brush-movement functions to maximise brush-visibility, that can easily be abstracted and used in different contexts.

## 6 Super-realistic Rendering

### 6.1 Introduction

#### 6.1.1 Definition

One of the main design considerations for all our renderers is real-time performance and we show in Chapters 3, 4, and 5 how non-photorealistic renderers can be designed and implemented to adhere to such rigorous constraints. It is only natural to investigate briefly the opposite and much more studied extreme of rendering. Photorealistic rendering is discussed in-depth in Appendix A and we note there that apart from very few exceptions (e.g. [101], [94] & [45]), true photorealistic rendering is not a real-time process, yet. Images, which are produced by modern computer games, include more and more realism-enhancing effects (like lens flares, shadow maps, light maps, etc.) but are still far from realistic. We therefore try to solve the problem of interactively rendering an object in real-time that is not only as realistic as can be expected by today's state of the art (i.e. computer games), but in fact truly photorealistic. We call this *super-realistic* and define that the resulting image quality should be comparable to that of video-footage.

### 6.2 Problems

#### 6.2.1 Problem Statement

We want our solution to super-realistic rendering to be sufficiently complete as to not be limited to a special subset of objects (e.g. only simple or uni-coloured objects). Specifically, we want to be able to render objects that have hair or fur or other features that are very difficult to reproduce using standard rendering techniques. As we stated above it is unreasonable to expect to find a solution to the problem in standard photorealistic rendering techniques like raytracing or the Radiosity method. We thus have to find a solution that

- circumvents common limitations in object complexity
- can be applied to a large enough group of objects
- can be implemented to perform in real-time

### 6.3 Solution

A very recent development in computer graphics, especially in connection with photo-realistic rendering is that of image-based rendering (IBR) (see [81], [57], [84], and [16] for applied examples). The idea here is that instead of creating all parts of a given scene from scratch, at least some of the scene elements can be re-generated by modifying and transforming appropriate sets of images. This is usually done for static scene elements like skies, horizons or other very distant scene elements (e.g. the panoramic images of QuicktimeVR: [3], [104]), but in fact even the very commonplace texture-mapping technique can be considered a very special application of image-based rendering.



In practical terms this means that if we have, for example, an airplane that flies through the sky, we do not necessarily need to render thousands of polygons, apply dozens of materials and calculate complicated light-interactions on every frame of our animation. It might be enough to render the airplane once and then just move an image of the plane through the sky – or better even, not render the plane at all but just take an image of a real one. Of course this solution is compelling, but what happens if the plane decides to bank or turn? In that case one image is not going to be enough, yet common sense will tell us that using one image for every possible viewpoint of the plane is simply not feasible. Our solution is therefore to limit the number of available viewpoints to a reasonable sample size and interpolate between samples. The following Sections show how this can be implemented in real-time.

## 6.4 Standard Approach

Let us consider the common affine transformations that one is likely to encounter in any typical graphics context (see Figure 72. Other transformations like skewing are acknowledged, but not considered typical):

- Translation
- Scaling
- Rotation

We will have to be able to apply all of these transformations to our object. Most of these transformations are easily accomplished by simply translating or scaling the image of the object and even a rotation in the image plane is trivial. Other rotations are unfortunately only possible for spherically symmetric objects. If we tried, for example, to rotate the airplane in Figure 72 about its longitudinal axis, we would expect the wings to foreshorten and disappear at 90 degrees. In general, object detail can appear and vanish under arbitrary rotation. We need to address this issue.

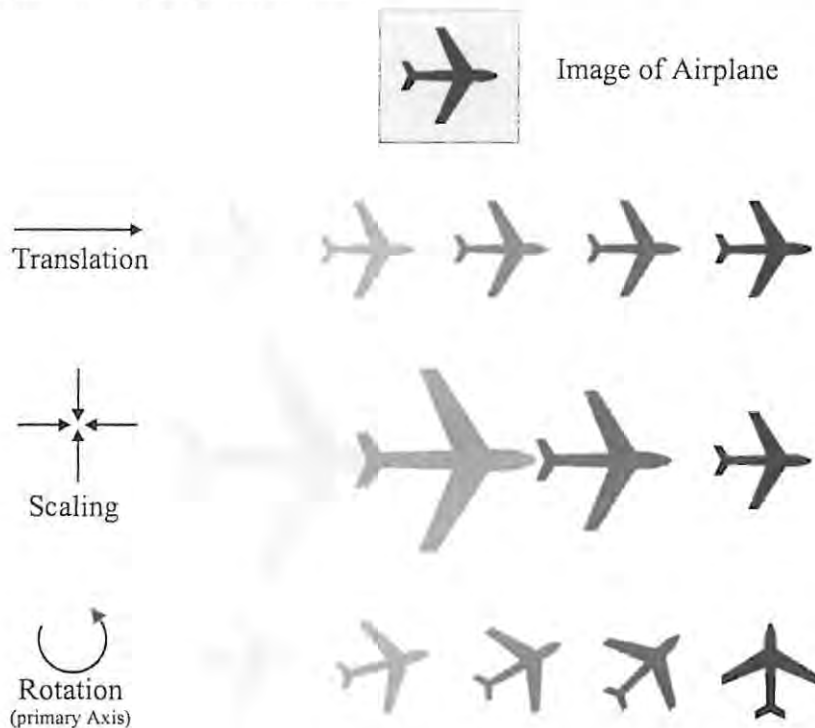
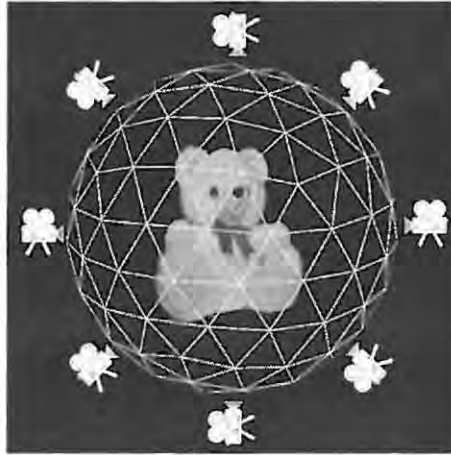


Figure 72 - Image-based rendering, affine transformations

To deal with this situation, we need more samples of the object, i.e. more images. As stated before, we cannot possibly take one sample per viewpoint, as there are infinitely many, so instead we opt for regular spherical samples as illustrated in Figure 73. At each vertex on the spherical grid we place a camera pointing towards the centre of the sphere and take a snapshot. The density of the spherical mesh then determines the number of samples to be obtained.



**Figure 73 - Geo-spherical sampling of a real-world object**

It should be obvious that the number of samples is proportional to the probability of finding a suitable sample for a given viewpoint. This means that we should have as many samples as possible. On the other hand, each sample takes up memory-resources, so we should try to use as few samples as possible.

Another observation is important. If a viewpoint happens to fall exactly on a sample-point (i.e. vertex on sampling mesh) then we can simply use that sample. It is much more likely though, that an arbitrary viewpoint will fall between sample-points and we have to deal with this situation. The solutions available are very similar to those of texture mapping (where sampling and aliasing are of equally great importance):

- Use the nearest sample (in terms of distance on the approximated sphere)
- Interpolate between several near samples

So far we have only found the first solution being considered [110] (QuicktimeVR), but we argue that the second solution is far superior. For a visual comparison, watch Animation H and Animation K.

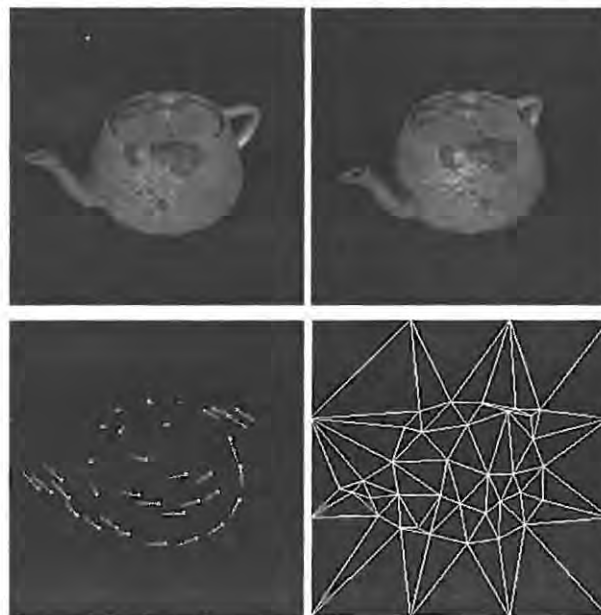
## **6.5 Optimisations**

### **6.5.1 Interpolation of Samples**

While the first solution is unquestionably the easiest, it will also produce the worst visual results: the object will appear to jump between orientations – smooth transitions are impossible. The second solution takes into account not only the very nearest sample, but several near samples. Since the mesh in our example is triangulated (see Figure 73), we use the three vertices of the triangle in which the current

viewpoint is located to interpolate a given orientation. We achieve this by using tri-linear or even higher order interpolation. For the sake of this discussion, we introduce a modification to our original set-up. We will first look only at rotation around one arbitrary but fixed axis. This is equivalent to spanning an arbitrary equator around the sampling sphere and placing samples exactly on the circle formed by the equator. This limits the first-order interpolation to simple linear interpolation, but everything else remains the same. We will then use bi-linear interpolation to improve on the interpolation result and show that the generalisation step from bi-linear to tri-linear interpolation is easily achieved.

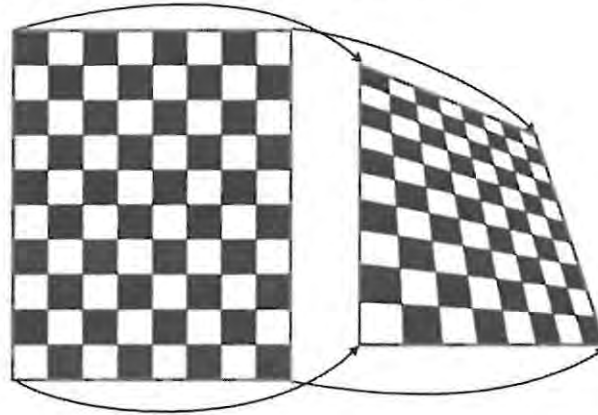
Our first attempt is to simply blend two images together. If we suppose samples are available at 15-degree intervals (i.e. 24 samples) and we want to simulate a view of exactly 20 degrees, we can mix 2/3 of the 15-degree sample with 1/3 of the 30-degree sample. The resulting effect is a much smoother transition between samples, but geometric detail still vanishes and appears from seemingly nowhere.



**Figure 74 - Teapot sampling: a) 15 Degrees; b) 30 Degrees; c) Control-Points and Motion Splines; d) Triangulation based on Control-points**

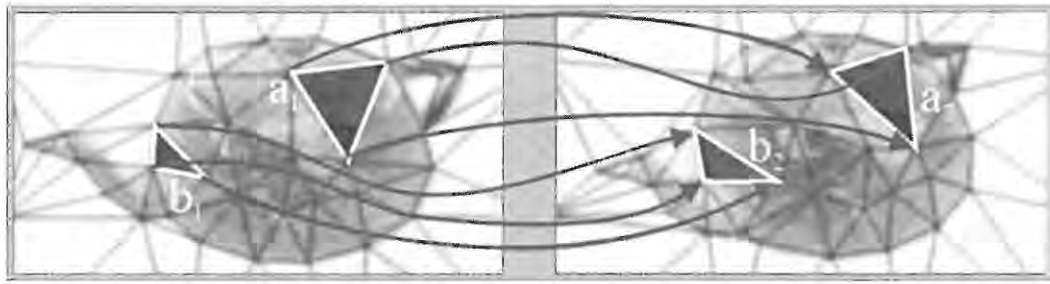
If we look closer at Figure 74a) and b) we understand the problem: The spout for example moves down and to the right, while the handle moves up and to the left. In addition to this the body of the pot hides parts of the handle. Mere blending cannot simulate this behaviour. We actually have to move the pixels corresponding to the spout to the correct location (and similar with all other features). It would obviously be totally unfeasible to specify the movement of all pixels between each pair of samples. On the one hand this would create another huge amount of information and on the other hand this information is simply not available for all pixels (as we mentioned above with the handle). Instead, we specify key-positions, called *control-points*, at easily identifiable positions in both images (e.g. the spout, the handle, the lid and various spots on the body of the pot). This is shown in Figure 74c). We then move control-points along a path from their starting-position in one image to their destination in the next. Figure 74d) shows how the control-points of Figure 74c) are connected using a triangulation scheme after Delaunay (see [106], [107] and [113]). Other schemes can be used as long as they guarantee an optimal triangulation (i.e. no

intersecting triangles) and offer fast implementations. The next question is how to move the remaining pixels around, so that we obtain a smooth transition between images. The answer in this case is as simple as it is effective: Texture mapping allows us to specify texture vertices (in our case these are 2D coordinates inside an image) that correspond to geometric vertices.



**Figure 75 - Pixel Interpolation through texture mapping**

Figure 75 shows two quadrilaterals, the one on the right being derived at by deformation of the left one. Several facts should be noted. Firstly, all we have done is move the defining vertices of the first quadrilateral around and have obtained what appears to be the same quadrilateral moved into the page and slanted slightly at an arbitrary angle. Of course the paper of this thesis is really flat and all we have done is move some points around arbitrarily, but our brain will tell us from past perspective experiences that we are looking at a very three-dimensional looking checkerboard. The habit of our brain to interpret two-dimensional movements as three-dimensional (if this movement conforms in any way with past experiences) is strongly exploited by our renderer. In fact all vertex-movement in our renderer is two-dimensional, but it complies with what we'd expect the corresponding three-dimensional movement to look like. This is what adds depth to our rendered images and makes them look convincing under animation. Secondly, we consider the checkerboard texture. To change one quadrilateral to the next, we did not have to specify the movement of all pixels in the source or destination images. Instead, we set up a rule of how the checkerboard texture is to be applied to the quadrilateral (this is done by specifying texture co-ordinates for each geometric vertex). This rule doesn't change when the quadrilateral is deformed and so the same rule applies, producing the deformed checkerboard on the right. In practice this means that by defining a texturing rule and the movement of deformation, we can implicitly specify the movement of all pixels contained within the quadrilateral. Lastly, we should mention that the quadrilateral shape was chosen for demonstration purposes only and the above discussion holds for any polygonal shape that texture mapping can be applied to. To see the teapot rotate from 15-30 degrees using our interpolation scheme, see Animation I (after watching several times, you might notice an artefact on the lid of the pot near the handle. The selection of control-points here was not ideal).

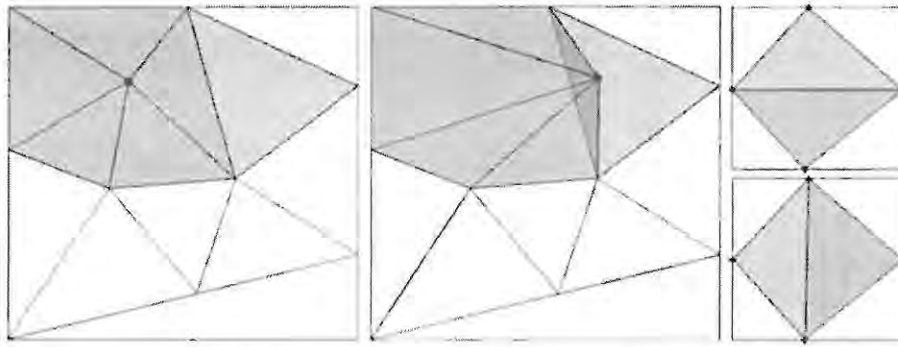


**Figure 76 - Linear interpolation in action**

Figure 76 shows how we use these ideas for our super-realistic renderer. The left image is an original and shows a teapot at 15 degrees rotation with triangulation applied. The right image is synthesised and depicts the same teapot at 30 degrees with the same triangulation-configuration (this means that the same vertices are connected by triangles, even though the connected vertices might have moved slightly). For example, the vertices defining triangle  $a_1$  are the same as those defining triangle  $a_2$  (and similarly with triangles  $b_1$  and  $b_2$ ), but these vertices have moved slightly between the two images. It should be noted that similar to Figure 75, the texturing information of triangles  $a_1$  and  $a_2$  is the same, so that all pixels contained in  $a_1$  are relocated to the corresponding location in  $a_2$ . This is important and it means that by deforming the vertices on the left to the vertices on the right, but using the same triangulation and texturing information, we can deform the teapot on the left to the teapot on the right. This process is what we call *linear interpolation*. If we now have another original image of the teapot at 30 degrees, we could easily reverse the process and try to synthesize the 15 degree teapot. It should be obvious, that interpolated images will look worse, the further they are removed from their source, i.e. if we have an original at 30 degrees, then interpolations at 29 and 28 degrees will look very realistic, while interpolations at 20 or 19 degrees will look a lot less realistic and at some stage the interpolation will break down completely. Fortunately, we have another original at 15 degrees, so it makes sense to interpolate the 15 degree original towards 30 degrees and the 30 degree original towards 15 degrees and blend the two results weighted by which side we are closer to. For example, for 20 degrees, we take 2/3 of the 15 degree interpolation and 1/3 of the 30 degree interpolation. This bi-directional process is logically called *bi-linear interpolation*. For a comparison between linear and bi-linear interpolation, study Animation L.

The visual result of this new approach is almost perfect: Image transitions are extremely smooth and object geometry moves instead of simply appearing. No additional information is necessary for this bi-linear interpolation, as control-points move along the same path, merely in opposite directions.





**Figure 77 - Triangulation Problems: a) Initial Triangulation; b) Control-point moves into neighbouring triangle; c) Two possible triangulations of the same control-points**

A visual artefact caused by the animated triangulation needs addressing. Triangles are computed based on control-point information of a particular sample. As the control-points move towards their destination location it can occur that triangles move into one another (see Figure 77a&b). Due to the necessary blending operations, this results in overlapping areas being over-exposed, i.e. too bright (Figure 77b shows the opposite effect for illustrative purposes). We found several solutions to this problem:

- Mid-point triangulation
- Per frame re-triangulation
- Depth-Buffering

Triangulation can be computed based on the mid-position of corresponding control-vertices instead of the start-position. While this lowers the probability of triangle overlaps, it does not guarantee to eliminate it.

Triangles can be recomputed on each frame. This guarantees that no triangles will overlap, but is fairly costly and results in yet another problem. Since any triangulation scheme has to decide how to connect neighbouring vertices and these decisions are based on the relative distances between neighbours, the connection layout can change instantaneously from one frame to the next. Figure 77c) shows two possible triangulations of the same vertices. Since the triangulation has a direct influence on the texture-mapping process, spontaneous triangle flipping will produce *visual clicks* (i.e. temporal incoherence).

The last solution uses the depth buffer to allow each pixel only to be written to once. As we have more than one render pass (2 for bi-linear and 3 for tri-linear interpolation), this means that the depth buffer has to be cleared after each rendering pass. This operation is also costly, but hardware-accelerated and therefore feasible. Alternatively, each rendering pass can be layered on top of each other with slightly decreasing z-value. While this is much more efficient, care has to be taken under perspective projection in order not to cause foreshortening between layers. This means that successive layers have to be very close to one another and depth buffer resolution has to be taken into account. Despite these issues, this method produces no visual artefacts like the other two solutions.

### 6.5.2 Control Vertex Acquisition

So far we have assumed the existence of suitable control-vertices for use in the triangulation process, but have omitted to reveal how these control-vertices can be acquired. This Section addresses this issue.

For testing purposes and adjustment of our main algorithms, we first selected control-vertices manually. While this proved to be extremely reliable, we quickly realised that this task had to be automated. Nonetheless we could identify from our own experience the basic steps necessary to gather control-vertices.

- Identify suitable control-Vertices (these include edges, contours and other recognisable landmarks).
- Find these control-Vertices in adjacent images

To implement the steps above we define the following set of rules that we adhere to in order to identify viable control-points. For this discussion we define a cluster of image-pixels as a *landmark* (in our case a rectangular area of pixels, the location of which is defined by its geometric mid-point). The rules are listed in descending precedence so that a short-circuit evaluation can be performed:

1. Landmarks may not lie too close to image boundaries (discontinuities)
2. Landmarks may not lie too close to one another (redundancy)
3. Landmarks must lie on regions of high frequency (it is implied that most of the landmarks are enclosed by edges)
4. Normalised Cross Correlation (NCC)<sup>5</sup> value of surrounding Landmarks varies significantly from Landmark under inspection (this makes a correct match in the adjacent Image more likely)
5. Not more than a certain number of NCC matches may be found (uniqueness)
6. { Landmarks can be found in the original and neighbouring image (self-verification) }

To explain some of these steps in more detail, we devised the demonstration shown in Figure 78. Image a) shows the original image and image b) depicts a modified but similar version of a) (as we would expect in any practical situation). Compared to image a), image b) is translated up and to the right, scaled down (shrunk in size) and slightly skewed at the top (i.e. top border is shorter than bottom border).

---

<sup>5</sup> NCC defines a window of pixels in source and destination images as well as a formula to determine the similarity of intensities in these windows. A perfect match scores 1, while 0 signifies no correlation. For more detail, see [65]

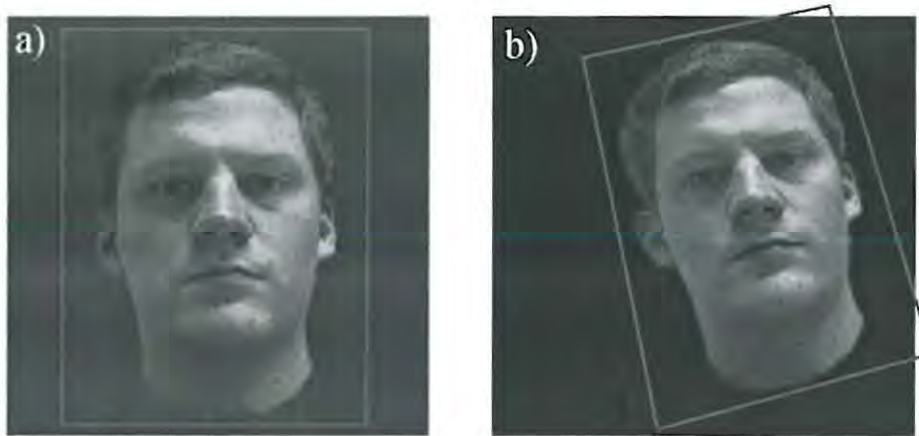


Figure 78 - Demonstration Set-up

The first rule is easily implemented by only starting to look for suitable landmarks a certain distance away from the image boundaries. To improve search performance, the second rule uses a temporary bitmap, into which circles are drawn for each valid landmark (i.e. flagging invalid positions). When traversing the original image for a suitable location, we simply check the current location in this temporary bitmap for validity. Rule three is enforced by performing an edge detection on the input image (Figure 79a) and only allowing landmarks to be placed on locations of strong edges. This is done to place landmarks along the contour of the object (to preserve its shape) as well as on important object detail (like the mouth, the nose, the eyes, etc.). We also assume that recognition of landmarks containing high-frequencies is easier than of those with low frequencies.



Figure 79 - Image analysis: a) Edge Detection; b) a possible landmark; c) many possible matches

The workings of the NCC algorithm can be seen in Figure 79b&c). We manually selected a landmark in the original image and tried to find the corresponding landmark in the adjacent image. The green dots show regions with a good NCC match, while the red dot shows the correctly identified landmark. This illustrates two facts. Firstly, in many cases the NCC will identify a very large number of probable matches, i.e. many parts of the images look alike and secondly, it is usually quite good at identifying the correct one within these matches. Of course the number of probable matches is also determined by the threshold value defining a *good match*. If too many probable matches are found, the landmark is not unique enough and we cannot assume that a correct match will be found. Rule five implements this notion. Finally, rule six states that if a landmark A was identified in the source image and found as landmark B in the target image, then the reverse should also hold, i.e. when searching for landmark B in

the original image, we should obtain the location of landmark A. In practice this rule may lead to too many false rejections and is very time-consuming, so that we do not make use of it.

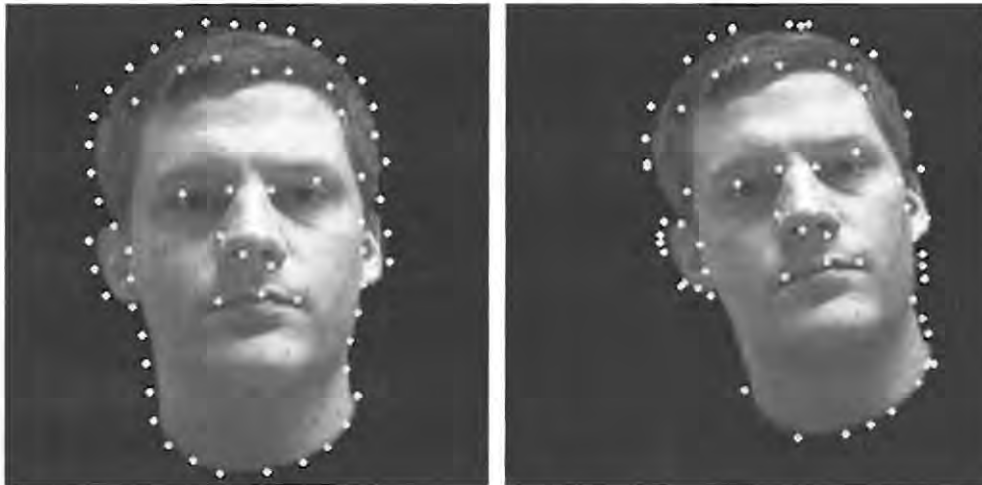


Figure 80 - Automatic Landmark identification: a) source; b) target

It should be obvious that a large number of variables have an influence on the landmark identification process (e.g. edge-detection matrix, NCC window size, minimum NCC score, maximum number of matches, etc.) and their tweaking is as difficult as it is essential. Figure 80 shows the result of a demonstration run. We can see on the left hand side that landmarks are indeed spaced evenly and located around the contour of the object as well as important features (eyes, mouth and nose, as well as parts of the hairline). This means that the first three rules are working as planned. Furthermore, it is evident that all landmarks inside the face are correctly matched. Unfortunately, a large number of landmarks on the contour are mismatched. If we consider Figure 81, the problem becomes obvious. Landmarks along the contour are not very unique. Figure 81a) marks the location of a proposed landmark. The naked eye has difficulty identifying any recognisable spot along the neckline. Figure 81b) verifies this fact, by marking the whole neckline as well as other parts of the face as possible match-sites. The actual match is incorrect, but impressively close to the correct location. In all likelihood, the variables determining the behaviour of rule four have to be adjusted. The obvious problem with choosing values for these variables is that they should be strict enough to produce good landmarks and matches, while on the other hand not being too strict so that not enough landmarks fulfil the criteria. The time taken to find and verify a suitable landmark is another important factor. For images of  $256 \times 256$  pixels<sup>2</sup>, and an NCC window-size of  $11 \times 11$  pixels<sup>2</sup>, we are able to match a given landmark within 440 milliseconds on our test-system. This is to say that a single landmark match through NCC takes 440 milliseconds, but this does not include the computation of other rules such as the edge-rule or the high-frequency-rule even though they have a very definite impact on the choice of landmark placement. To demonstrate the fragility and sensitivity of the variables used in our automatic landmark identification and verification process, consider the following: the results of Figure 80, i.e. 65 possible sites for landmarks along with matches in the neighbouring image (about 10% of which are incorrect) were obtained within 135 seconds (i.e. over two minutes). These results were obtained after a slight adjustment in one of the algorithm's variables (the exact variable and amount are irrelevant here). Before this adjustment, we obtained a mere 15 landmarks, only 10 of which were correct in 969 seconds (i.e. over a quarter of an hour). While the exact numbers are of no real



concern, it should be noted that automatic landmark identification and verification is a non-trivial and very time-consuming task.



**Figure 81 - Contour problems: a) proposed landmark; b) possible and incorrect matches**

Another problem area is that of appearing and disappearing features of an object. These need to be carved out of adjoining parts of the object, but by definition have no corresponding equivalents in neighbouring images. These issues are part of an ongoing investigation into the matter. One possible direction that we will be investigating is to use spatial relationships of established landmarks in order to place and identify additional ones in a boot-strapping process. This means that if we place a landmark between two established landmarks in the source image, we can reasonably expect the equivalent landmark to be situated in between corresponding landmarks in the target image. This might especially improve the placement of edge-landmarks for which shape, location and contour are better distinguishable than NCC values.

Another possibility of selecting control-points, though not always applicable, is available if the geometric shape of an object is known. The teapot in Figure 74, for example, has been rendered by a ray-tracer using a 3D-object file. If this kind of information is known, we can easily deduce the necessary control-points to perform the interpolation. It is obvious that this is rarely the case, when taking pictures of live elements like faces or people etc. Nonetheless crude 3D replicas could be constructed from which the control-points can then be derived.

Motion detection schemes are used in video compression algorithms like MPEG-Video [109]. These compression techniques rely heavily on inter-frame coherence and will try to find already encoded areas of one frame in the following frame. If such an area can be found, then only the motion-vector needs to be encoded instead of the area itself. Unfortunately the task here is actually easier than in our case: Since the whole image needs to be encoded, each area of the reference image is compared to the successor image. Our method on the other hand requires identification of very few but important landmarks. After all, our main problem is the *identification* of such landmarks and not their discovery in subsequent frames. In addition to this, the video compression technique does not need to be concerned with the content of the



encoded images. If we consider a very reflective teapot, then the video compression scheme could get away with interchanging the real spout with a reflected spout and vice versa (this is because we only see two successive frames and nothing in between). In our case on the other hand, we are faced with several derived frames and so the movement of the real spout to the reflected spout and vice versa would become evident. This means that standard video encoding techniques are rather unsuitable for our purposes.

### 6.5.3 Generalisation to 3D

As promised above, we will take a moment and consider the implications that our simplifications have on our solution. We have discussed linear and bi-linear sample-interpolation along an arbitrary equator. In the more general case we will not move along a line or circle with evenly spaced samples, but on a surface. The difference is that instead of exactly two nearest neighbours, we will, in general, have more than two. As we chose to triangulate the surface of our sampling sphere, we are usually faced with three nearest neighbours as shown in Figure 82.

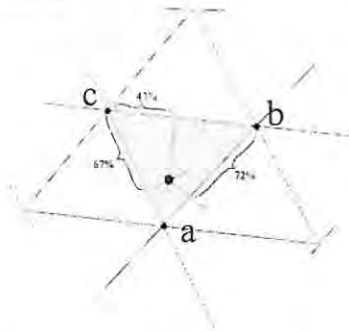


Figure 82 - Tri-linear sampling for arbitrary rotations

This means that instead of the bi-linear interpolation we perform, we have to use tri-linear interpolation to achieve the best visual result. Figure 82 shows a typical set-up: On the surface of the mesh, the current viewpoint (black dot) is somewhere on an arbitrary triangle (to visualise, our examples so far dealt with view-points on mesh edges). As each pair of neighbouring sample-points has its own transformation information, we basically have to perform three linear interpolations (indicated by the intersections of the dotted lines with the mesh edges) and blend them accordingly. Even by tripling the computations of each frame, we can still perform well inside real-time constraints.

### 6.5.4 Cubic Interpolation

So far we have only considered multi-linear interpolation. In most cases linear movement of object features does not correspond well to the natural motion expected from an object under rotation. To address this issue, we simply use cubic splines along which the control-points can move. The added computational cost is minimal and the visual improvement justifies it easily.

## 6.6 Extensions

Even though we designed our super-realistic renderer to be integrated alongside several other renderers in a VR context, we found that it implements two processes called *morphing* and *tweening*. Even though these terms are often used interchangeably in the literature, we make the following distinction:

- *Tweening* – process of deriving intermediate frames in order to convert an object or scene in one key-frame into a **similar** object or scene in an adjacent key-frame.
- *Morphing* – process of deriving intermediate frames in order to convert an object or scene in one key-frame into **another** object or scene in an adjacent key-frame.

In effect this means that neighbouring images for tweening will have similar content, while those in morphing will generally have different content. This means that our super-realistic renderer is based on the concept of tweening, but we show in Animation M how morphing can be achieved just as easily and with exactly the same performance. As we have shown how to implement both of these techniques in real-time on home-PC level hardware, various industrial and commercial applications spring to mind.

### 6.6.1 Motion Pictures and Special Effects (Morphing & Tweening)

Many modern Motion Pictures and high-end commercials use computer special effects to draw audiences and make the seemingly impossible a reality (at least on Celluloid). Movies where morphing and/or tweening have been extensively used include *Terminator II* and more recently *The Matrix*. Even if the texture-size and image quality of most 3D accelerator cards cannot compete with production quality standards, we certainly believe that our system can aide in previewing and scene visualisation.

### 6.6.2 Web Advertisement (Tweening)

The idea of using image-based rendering for product advertisement on the web is not new [110]. A site-visitor can get a very realistic impression of the look and design of a certain product as she can interactively rotate it and view it from different angles. So far this is only done using the nearest sample technique. One might be tempted to think that this restriction is linked to bandwidth limitations. The information necessary to tween between each neighbouring pair of images is comprised of one motion-spline per control-vertex, i.e. three 2D co-ordinates inside the image per cubic spline. If we limit ourselves to reasonable image-dimensions, we need a maximum of 16 bits per position, i.e. 12 bytes per control-vertex. In our collection of tween-samples we register an average of 65 control-vertices (being part of an average 114 triangles), resulting in roughly 800 bytes of tweening-data. Compared to the uncompressed size of our sample-input images of 192Kb this additional bandwidth is negligible. This means that from a networking point of view our Super-realistic renderer is actually very efficient.

### 6.6.3 Extreme Low Bandwidth Video Conferencing (Tweening)

We studied a system in which our renderer is used to re-construct facial expressions [114]. In this system, a camera captures the image of a person. Using image analysis, the positions of eyebrows, lower eyelids, nose, mouth and chin are determined and used for the selection of a corresponding facial expression

(which include grimaces like laughing and frowning, but also mouth-positions when speaking certain phonemes). The code for a certain facial expression is transmitted over the network to the remote side for reconstruction. The remote side is initialised with an image of a neutral facial expression, which is deformed according to expression templates. Several points should be noted with respect to this application:

- Reconstructed images cannot correspond 100% with original images (features such as smile dimples, frown lines etc. cannot be reconstructed)
- Reconstruction is very easy and fast with our method
- Expression evaluation may be relative expensive depending on the complexity of the image analysis, but has been shown to work in real-time [65].

#### **6.6.4 Facial Character Animation (Tweening)**

Another application concerns the animation of virtual characters. In many cases some form of motion-capture (MC) is used to give characters a more lively appearance. MC can be performed amongst other methods using radio-trackers or video-evaluation but is in most cases expensive and time-consuming. This is especially true for facial expressions since, if the expression is formed on a purely geometrical level, many vertices have to undergo transformation and therefore have to be motion-captured.

We on the other hand believe that if the facial expression is performed at texture level (as with the Video Conferencing above) we can map an entire expression on a much less detailed and much less animated geometry, while still being able to smoothly vary between expressions or even combine them [56]. A demonstration using linear and bi-linear interpolations is available in Animation L. Animation J shows a full-body animation.

#### **6.6.5 Restoration of Video Material (Tweening)**

When capturing live video material to be used in our animation sequences, we came upon another possible application. Supposing video or film material in general is damaged in such a way that many frames are missing or of poor quality while other frames are of acceptable quality. We can then try to interpolate the healthy frames using tweening to reproduce the missing or damaged ones. We show this in working in Animation N, where the right hand side shows the 8 frames of an original video source (temporal and spatial resolutions intact) which might have been damaged, while the left hand side shows an interpolation of the first and last frames of the original source (all other frames could have been lost). We can see that the temporal resolution is greatly improved compared to the original (we are able to derive as many in-between frames as desired), as is the spatial resolution (automatic hardware-accelerated bi-linear filtering through texture smoothing). We also acknowledge that unique information that was contained in the original frames (e.g. the shadow moving over the doll's face) cannot be reproduced with our method, because no information of it is contained within the first or last frame.

## 6.7 Results

### 6.7.1 Comparison of Approaches

To measure the performance of different approaches and settings of our super-realistic renderer, we tweened samples with a resolution of  $256 \times 256$  pixel<sup>2</sup> and 24 bit colour depth onto a screen area with the same dimensions (this was done to better compare input vs. output quality and is unbeknown to the renderer, i.e. this is not an optimisation). Table 20 lists the relevant results. Firstly, the conventional approach in the top-most row simply renders a single texture-mapped quadrilateral and therefore performs the fastest with 660 frames per second. Our first approximation (linear interpolation without retriangulation, 2<sup>nd</sup> row) renders an average of 64 triangles at about 570 frames per second. Re-triangulation per frame (3<sup>rd</sup> row) severely impacts performance and drops the frame-rate to 167. Adding a second rendering pass to implement bi-linear interpolation (4<sup>th</sup> row) does not have a big influence on this, as the difference is only 3 frames per second. This indicates that the retriangulation is considerably slower than the rendering of triangles and we therefore strongly suggest to use either of the depth-buffer solutions discussed in Section 6.5.1 to solve the overlapping triangles problem. Our suggested setting in row 5 using bi-linear interpolation and layered rendering performs at 450 frames per second. The alternative depth-buffer solution (6<sup>th</sup> row) is slightly slower at 423 frames per second.

	Depth-Buffer Clear			Re-Trig per Frame		
					660	Nearest Sample approach
		*			570	Linear Interpolation (-R)
		*		*	167	Linear Interpolation (+R)
*		*	*	*	164	Bi-Linear Interpolation (+R)
*		*	*		450	Bi-Linear Interpolation (L)
	*	*	*		423	Bi-Linear Interpolation (D)

Table 20 – Comparative Results for Super-realistic rendering

These results show conclusively that super-realistic rendering is not only feasible, but presents a very fast and powerful image-based rendering method, capable of producing great visual detail with minimal computational overhead.

### 6.7.2 General comments on Factors influencing Results

#### 6.7.2.1 Quality

The single most important factor in terms of rendering quality is the number of samples that are available for interpolation. The more samples are available the better the output, because less guesswork has to be performed in terms of what happens in between samples.

Image quality is another obvious factor. The quality of the output images is limited by the quality of the input images. Since texture-interpolation and blending operations are applied to the input images, they should exhibit a reasonable resolution with respect to the expected screen-dimensions of the output. As a rule of thumb, the input images should not have a considerably lower resolution than the anticipated output image.

We have discussed the use of blending operations and multiple rendering passes. While these help to improve temporal coherence, they tend to blur the output and a compromise will therefore have to be made between temporal and visual quality.

#### 6.7.2.2 Performance

Since real-time considerations were a main aspect of the design, we pause briefly and investigate the performance issues involved.

Firstly, image acquisition, image evaluation as well as definition of control-vertices are non-trivial tasks and computationally expensive, but can all be performed off-line, before the actual rendering. At run-time the following tasks have to be performed:

- Translation of image position and rotation to a corresponding set of neighbouring Images.
- Depending on the quality of output needed, linear, bi-linear or tri-linear interpolation of neighbours; the speed of which mainly depends on the hardware-capabilities of the system.

Depending on the way the images are stored, the translation step is largely trivial. In most of our examples, we are dealing with about 30-80 triangles (a fairly low number considering the processing power of most modern 3D accelerators), which have to be rendered twice for bilinear interpolation. Even for three render passes, this can easily be performed in real-time.

One important consideration remains, which is the one of texture memory. Even though we can drastically reduce the amount needed for our renderer by using an interpolation scheme, we may still fill up texture memory rather quickly. This in turn may lead to thrashing if the amount of memory is too small. Possible solutions are:

- Using hardware texture compression offered by many modern graphics cards
- Taking advantage of object symmetry
- Decreasing degrees of freedom of object or observer (e.g. if a VR user is limited to wander the surface of a flat world, the possible elevation range towards the object is limited as well)
- Increasing interpolation span (which will in most cases result in loss of animation quality)

### 6.8 Summary

In this Chapter we have investigated the opposite extreme of non-photorealistic rendering, which we dubbed *super-realistic rendering* and define as “interactive rendering in television quality at real-time



frame-rates". Image-based rendering, which forms the basis of our solution, is briefly introduced. We then identify the affine transformations that need to be implemented in order to create a generally applicable renderer. Most of these transformations are trivial except for the general rotation. This problem is addressed via geo-spherical sampling and sample-interpolation. The interpolation scheme we developed is discussed in detail and related problems such as triangle overlapping are addressed successfully. We continue by investigating the issues of control-vertex identification (which still needs further research) and demonstrate the generalisation from planar rotations to arbitrary paths on the sampling sphere. We discuss a further optimisations in the form of expanding the linear interpolation to a cubic one. Following this, a variety of extensions in the form of possible applications are discussed. We argue network efficiency issues by stating that in average a mere 800 bytes of data are necessary to interpolate between two neighbouring sample-images. Finally, we prove that real-time performance of well above 400 frames per second is easily achieved even for high-quality multi-pass blending schemes and list the main factors influencing visual quality and performance.

In summary, we successfully implemented a super-realistic renderer which is capable, in real-time, of producing interpolated images of highest quality to simulate 6-degree-of-freedom-transformations, including convincing rotations.

## 7 System Integration

While a variety of topics have been discussed pertaining to the specifics of different renderers and their optimisations, there are further considerations to be entertained with respect to the hardware and software architecture used for our implementation. Our design choices are mostly influenced by real-time constraints, but a great emphasis is also placed on portability and system independency. System integration issues relating to performance as well as portability are discussed in this Chapter.

### 7.1 Software Considerations

As with all real-time systems, all software components (from the Application layer down to the Operating system and driver layers) are considered critical and have to interact smoothly in order to achieve optimal performance. The following Sections describe our design choices starting at the Application layer and work their way down to the low-level software components.

#### 7.1.1 Virtual Reality API

The following design aspects were critical in the choice of a suitable VR API:

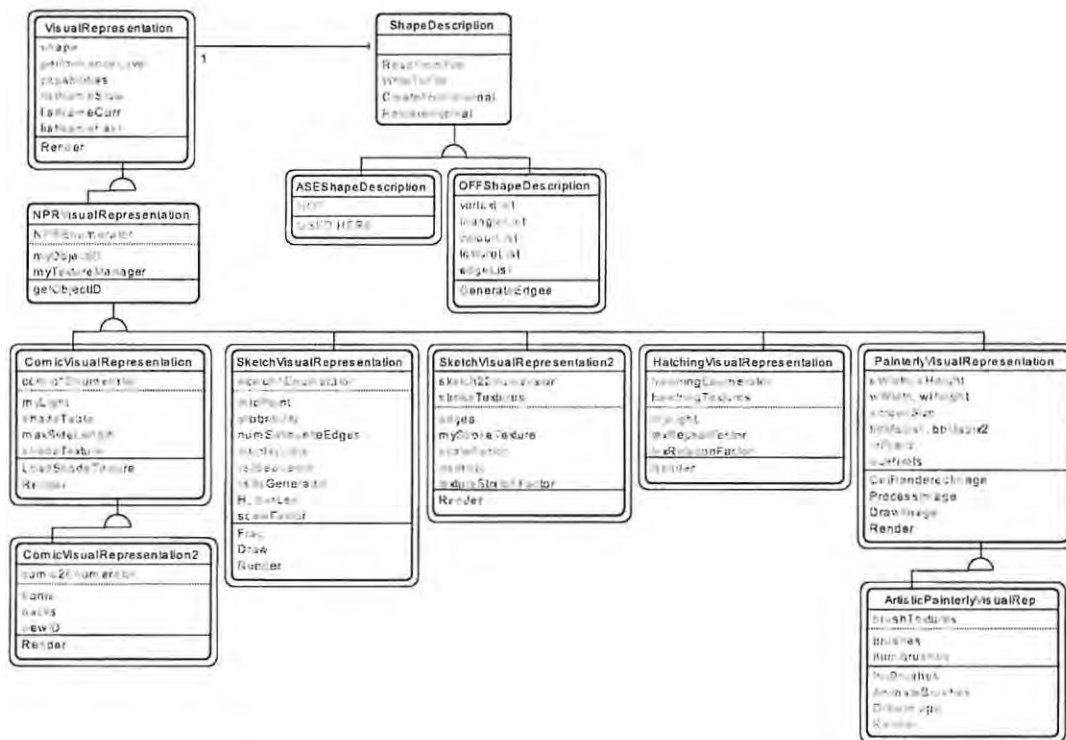
- Speed
- Flexibility
- Portability

And more specifically, the separation of object data from its visual representation.

All of these points are satisfied by CoRgi [112], a virtual reality API which is being developed at Rhodes University and which by now is released in its fourth generation. It is written in C++ and makes use of all the powerful features of object oriented programming to ensure the best possible combination of flexibility and high execution speed. Portability is encouraged and supported by encapsulating hardware and operating system dependencies in so-called *device* classes. These are usually highly platform dependent, while derived classes can rely on the common interface to base classes and are almost totally platform independent. This level of hardware abstraction allowed us to port the CoRgi system completely from a UNIX/IRIX environment to a Microsoft Windows environment with minimal effort. Conditional compilation and the use of compiler macros allow one and the same code to be compiled on a large variety of platforms while ensuring native code execution speeds.

The present object model of the CoRgi architecture supports different renderers to represent the same object data without change. The relevant parts of the object model are shown in Figure 83. While the abstract class *ShapeDescription* supplies the functional interface to the object data, the derived classes *ASEShapeDescription* and *OFFShapeDescription* provide format-specific data members and additional related functionality. This group of classes is thus responsible for loading object data in different formats (only the OFF implementation is shown) and for storing it internally. The next group of classes is

responsible for displaying this data and have as their parent the `VisualRepresentation` class. This class is not abstract and in fact implements a standard renderer. It knows about various capabilities (like smooth shading, lighting, double-sided rendering and the like), which can be enabled or disabled in order to achieve a certain performance level (this is to ensure a specified minimum frames per second count). Furthermore, three different displaylists can be defined for three levels of performance. Depending on the implementation these can differ in object resolution (i.e. by sub-sampling the object data) or in the capabilities enabled. The `Render` function of this class is defined virtual and has to be defined by any derived class. It is this function that performs the actual graphical rendering of any object. The `NPRVisualRepresentation` extends the functionality of the `VisualRepresentation`. It provides a common constructor to all the derived classes, but does not define a `Render` method (which is left to the inheriting classes).



**Figure 83 - CoRgi Object Model (Relevant Part)**

From the `NPRVisualRepresentation` we derive all NPR renderers. As can be seen in Figure 83, various classes have class variables (in the member variable Section above the dashed line) called `enumerators`. These ensure that class-resources like textures and tables are only allocated and initialised once (by the first object of this class) as well as deallocated properly (by the destructor of the last object of a given class). It should be noted that not all semantically related classes are necessarily derived from each other. For example, the `ComicVisualRepresentation2` is derived from the `ComicVisualRepresentation`, as it provides a graphical extension to the parent class. The sketching classes on the other hand are so different from an implementation point of view that they are not interrelated (apart from having the same parent). The `PainterlyVisualRepresentation` provides all the necessary functionality to pre-render a given object in default style (by calling the `VisualRepresentation's` `Render` method), capture it into a

memory-buffer, process the buffer information and re-render it via the `DrawImage` method. This design allows us to derive the `ArtisticVisualRepresentation` class (using brushes) from the `PainterlyVisualRepresentation` class and only override the `DrawImage` method (which becomes the equivalent `Render` method of the `Painterly` classes).

This object model allows us to share object data between `visualRepresentations` (i.e. one and the same object could be rendered in a variety of ways, without the need to have multiple copies of the object data committed to memory) in a flexible and efficient way. Care is taken to limit the necessary amount of resources by making use of class variables guarded by enumerators.

### 7.1.2 Graphics API

At the moment (due to a welcomed level of standardisation, powered to a large extent by the gaming industry) there are two commonly used 3D graphics API's: Direct3D by Microsoft and OpenGL (originally developed by Silicon Graphics). The functionality of these two API's is so similar that tools exist to emulate one via the other (e.g. [82]). Many modern games and applications allow the user to choose the API to perform the rendering. Availability of drivers for the two API's and their implementation might influence the choice in this case. As the Direct3D API is only available on Microsoft operating systems, we chose OpenGL as our implementation graphics API to ensure the greatest level of platform independence.

### 7.1.3 Operating Systems (OS)

Commonly used operating systems today include the Windows-family, Unix-family and Mac-OS-family. Due to the prior choices of programming language (C++) and graphics API (OpenGL), both of which are available on all of the above-mentioned operating systems, we ensure the greatest possible platform independence in combination with the highest possible performance. Our system has been compiled and tested successfully on SGI stations running IRIX, as well as Intel-class machines running LINUX, Windows 98 and Windows 2000. While certain low-level object classes including devices and GUI elements have to be re-written for every operating system, the render-specific classes dealt with in this thesis (namely the `visual Representations`) were converted without changes due to the operating system. The components of an OS which have the greatest influence on graphics performance are:

- Graphics Driver - A bad or missing implementation might mean falling back on software emulation of features that are generally hardware-accelerated
- Multi-threading – Even though not currently enabled, the CoRgi system introduced in Section 7.1.1 is in essence multi-threaded, allowing for decoupled processing of individual system components. The OS has to actually support multiple threads in order to take advantage of this feature.
- Multi-processor support – The ability of running different parts of a rendering system on separate CPUs can improve overall performance by lessening the load on each individual CPU. Profiling of our demo programs showed, for example, that Windows 2000 will run our main

execution module on a different processor to the OpenGL library. Not all OSs support multiple processors (e.g. Windows 2000 does, while Windows 98 does not).

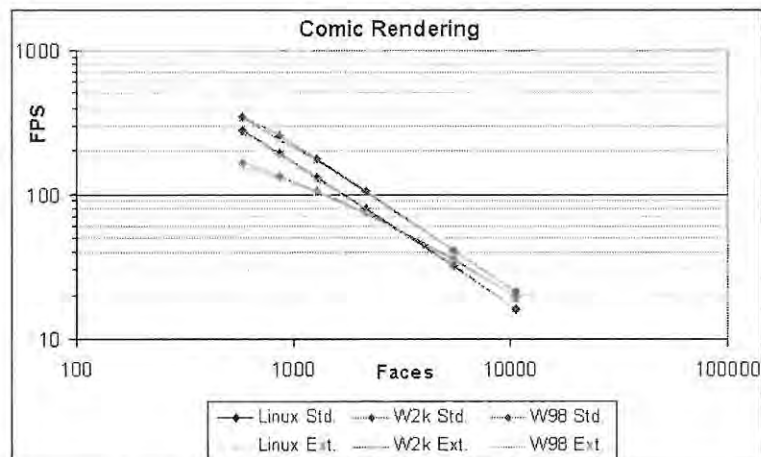
- Others – Resource management and security policies also have a logical impact on program execution, but their impact is considered small compared to earlier points.

To show how our renderers perform on various operating systems, but without going into unnecessary detail, we performed a series of tests on a single machine with three different operating systems installed (a more thorough profiling of code-segments on various machines was also carried out, but an appropriate discussion would distract from the main technical rendering issues. Relevant figures can be obtained from the accompanying CD or the author if required). Table 21 details the test configuration.

Hardware			
CPU	Dual Intel Pentium III @ 500Mhz each		
Memory	512 MB		
Graphics Card	Asus GeForce2 GTS		
Harddisk	Quantum Fireball lct10 15Gb		
Software			
OS	Linux Redhat 7.1	Windows 2000	Windows 98 SE
OpenGL	1.2	1.2	1.3
Graphics Driver	NVidia Reference 12.51	Asus Unified 5.33	NVidia Reference 21.85

**Table 21 - Configuration for multiple OS Test**

The tests were performed exactly as described in Section 2.4 and we list the relevant results here, grouped by rendering style. Linux results are always blue, Windows 2000 (W2K) results are pink or red, and Windows 98 (W98) results are green. The same rendering styles on different OSs have the same markers. It should be noted that the test-machine has two processors installed, a fact of which only Linux and W2K can take advantage.



**Figure 84 - OS Comic Performance**



Figure 84 shows results for the comic renderers (Std. = Standard; Ext. = Extended). The first interesting fact is that the performances of the two renderers are so close on all of the OSs that we had to apply slightly brighter colours to the Extended comic renderer results to make them distinguishable. This conclusively proves the validity of our various optimisations that enable us to render a more sophisticated lighting model in the same time as it takes to render a basic lighting model without these optimisations. The second point of interest is that all OSs perform fairly close to one another and that the dual-processor capabilities of Linux and W2K do not seem to have much an effect on rendering performance, otherwise we would expect more of a gap between them and W98.

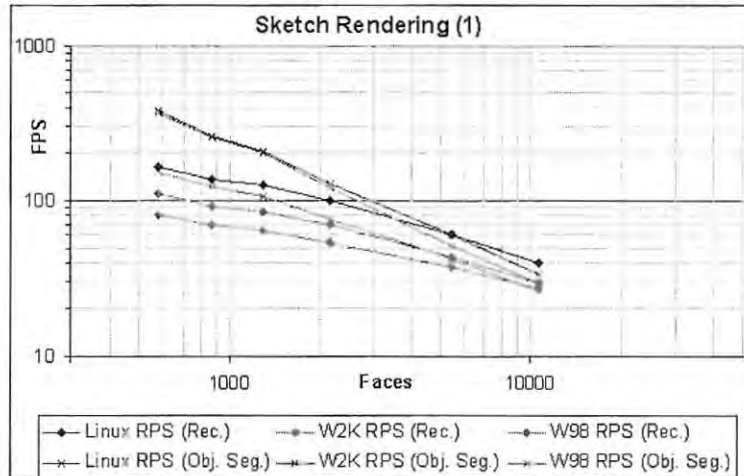


Figure 85 - OS Sketch Performance (1)

Due to the large number of implemented sketch renderers and approaches, we list the results in two parts. The first part, displayed in Figure 85, groups together the outlining sketch renderers. As expected, the object-segmentation approach (Obj. Seg.) outperforms the recursive approach (Rec.) throughout. An interesting result is that Linux stands the clear winner in this category, closely followed by W2K and further afield, W98. We attribute this to what we believe to be superior displaylist performance under Linux, which these renderers make extended use of.

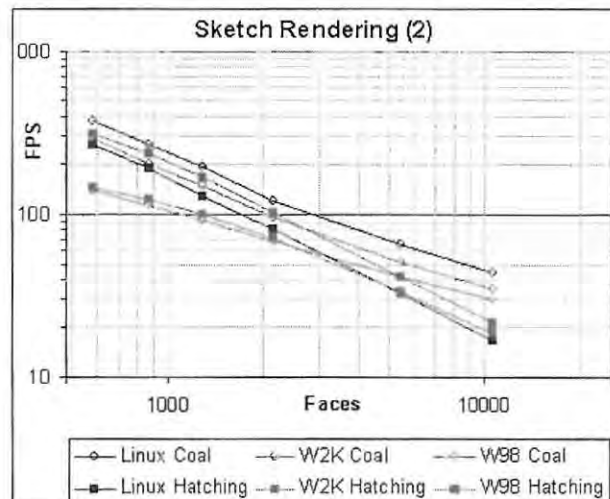
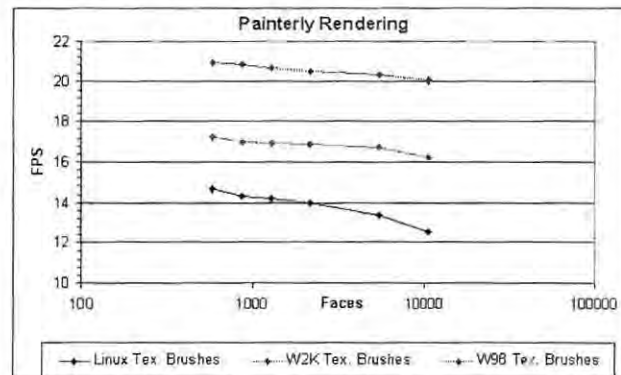


Figure 86 - OS Sketch Performance (2)

Sketch renderers using texturing as their main drawing element are tested in Figure 86. Both coal and hatching styles lie closely together under Windows OSs, but Linux fares much poorer for the hatching renderer than for the coal renderer. Even though this does not become obvious from Figure 84, because the test-objects do not have base-textures, we have encountered similar situations that are suggestive of the fact that multi-texturing, used in the hatching renderer, is not well supported in the Linux graphics driver. Again, W98 performs the poorest.



**Figure 87 - OS Painting Performance**

Performance values for painterly rendering are shown in Figure 87. As expected, the number of faces has no large impact on rendering, which is mainly influenced by the number of brushes (in this case 4000). Surprisingly, Linux fares worst this time instead of W98, which comes in second place after W2K. This is especially troublesome, as single-texturing results for the comic and sketch renderers under Linux are equivalent to those for W2K, so that we fail to find an obvious explanation. Still, it is pleasing to see that refresh-rates of over 10 frames per second are achieved under all OSs.

Apart from listing performance results for three different OSs, this Section conveniently brings together performance figures for all our common NPR style implementations. One important result of this investigation is that all our renderers perform between 12 and 370 frames per second for our selection of OSs. Another, very interesting conclusion can be drawn by comparing Table 1 and Table 21, which list the configurations of our different test-systems, in connection with the results obtained in this Section and the individual result Sections of the NPR renderers. While the system used in this Section has double the number of processors which are Pentium III chips instead of Pentium II, 320MB more of RAM, and a more advanced graphics card (GeForce2 instead of GeForce), our other system (used for results in Chapters 3, 4, 5, and 6) performs extremely competitively and in most cases even outperforms the better-equipped system. This we attribute solely to finding the graphics card driver which best suits a given graphics card, which we spent much more time on for our smaller test-system. Different drivers and even different versions of the same driver can perform vastly different for the same card but different vendors, i.e. an NVidia reference driver might perform perfectly well on a Creative Labs GeForce, while producing poor results on an Asus GeForce. Ensuring that all components of a system work together smoothly and efficiently can be more advantageous than simply buying faster or better components.

## 7.2 Hardware Considerations

The hardware used to run our system is of great importance to its performance. The hardware components most influential are:

- 3D Graphics Accelerator
- Processor Speed
- Main Memory

Various factors determine the execution-speed benefits that can be derived from a 3D graphics accelerator. On one level there are the capabilities of the device. While most of the graphics commands used in our renderers are so standardised that they can be considered guaranteed on all graphics cards, there are other commands (like the multi-texturing used in the extended comic renderer) which are not implemented by all vendors. This does not mean that these commands or the resulting functionality cannot be emulated in software (i.e. by the CPU of the host computer) or by performing multiple rendering passes (i.e. rendering a scene or elements thereof several times and accumulating the effects using blending or the accumulation buffer), but emulation always results in a rather heavy performance loss. In principle a 3D accelerator card is not needed for any of our renderers, but real-time responses cannot be expected without one. On another level the performance of the graphics card itself has a drastic impact on the rendering performance of the system as a whole. The number of basic 3D elements (usually triangles) that can be rendered per second as well as the fill-rate (screen-elements, i.e. pixels) are of great importance here. In addition to this, more and more elements of the rendering pipeline are emigrated from the host computer onto the graphics card. The latest generation of graphics cards for example even performs the transform and lighting calculations on-board (a task which previously was performed by the host CPU. For a technical brief on this topic, see [60]). The result of this development is that the host system is becoming less important for the actual rendering process (in reality we can expect that the host CPU will be more involved in other tasks like game AI etc.).

On systems where operations such as transform and lighting still are performed by the host CPU, its speed is important to ensure that data in the graphics pipeline can flow smoothly. In addition to this other calculations (like customised lighting, edge-detection, etc.) may have to be performed on top of the rendering calculations. A powerful CPU is therefore an advantage.

Main memory is only a minor factor. As with all applications, our system runs faster and smoother if enough main memory is available to hold the entire program and associated data in memory as opposed to swapping it out onto virtual memory. A more important factor is actually the memory on the graphics board. Apart from having to contain various full-screen buffers (usually RGBA and Z, but in some cases also stencil or off-screen), the graphics card has to hold the textures to be mapped onto graphical elements. If texture memory is limited or too many or too detailed textures are used, these textures will be swapped out to main memory. This process is costly and significantly affects rendering performance. The negative effect of this can somehow be limited by using an AGP graphics card with the maximum bus-

bandwidth available. Texture-compression methods are also available. The number and size of our textures are limited though, so that these factors should not have a great effect on the rendering performance.

### 7.3 Availability issues

Availability is a very real concern with respect to our renderers as we try to provide solutions that are widely applicable. Examples of features which are known to vary widely either in their actual availability or implementation include:

- Multi-texturing (number of supported units)
- 3D texturing (hardware and driver support)
- Auxiliary Buffers (availability)
- Stereo Buffers (availability)
- Pbuffers (availability and driver support)

In all cases where availability is questionable, we provide alternative solutions, which will produce equivalent visual results, usually at the cost of decreased performance. It should also be noted that even though certain features might be available it is not always guaranteed that they are optimised in hardware. On our test-system for example both 2D texturing and 3D texturing are available, but only 2D texturing is hardware-accelerated. As hardware-acceleration has a great impact on rendering performance, this issue needs to be considered.

### 7.4 Summary

In this Chapter we examined performance and portability issues relevant to our NPR renderers. We introduced and discussed our choice of VR API, CoRgi, into which we plug our renderer classes. An object model of the visual representation classes within this API is delivered, along with pertinent member variables and functions.

Our motivation for choosing OpenGL as our implementation graphics language follows.

Operating systems, along with a limited discussion of their effect on our renderers, are also studied. Several interesting results emerge, the most important of which is that all our NPR renderers perform above the specified 10 frames per second limit on any operating system and in most cases even in the hundreds of frames per second. Secondly, our test-hardware for this series of tests being far superior to that used in the rest of this thesis does not imply a necessary improvement in performance. This shows that hardware-dependence is limited and implies that our renderers will perform very well on most modern systems with a hardware-accelerated graphics card.

Further hardware issues are addressed in the next Section and it is our finding that the graphics card, more than CPU speed or host memory, determines rendering performance.

Availability of certain graphics features like multi-texturing, 3D texturing or various extension buffers, is not guaranteed and we discuss related issues next. We find that while performance is usually negatively affected, most unavailable features can be emulated using available ones.

In conclusion, we look into both software and hardware dependencies of our NPR system, list the available design choices and motivate our decisions. We perform a successful series of tests that tie together the performances of all our common NPR renderers on an alternative testing platform to that used in the rest of the thesis, the result of which is that real-time performance of our NPR enhancements is not limited to one particular test set-up, but can be replicated on different hardware and different operating systems.



## 8 Conclusion

When work began on this thesis about two years ago, *NPR rendering* had just started to make a name for itself and was starting to appear in the proceedings of well-known conferences. *Real-time* NPR rendering was mostly still unheard of, but the raw computational power of modern graphics cards was highly suggestive that this notion could be entertained. We therefore set out to investigate how common NPR styles could be implemented to adhere to rigorous constraints with respect to both *performance* and *quality*. In practical terms this meant that our renderers would have to perform interactively in real-time (at least 10 frames per second) and that the visual quality be as convincing (resembling real-world examples of the attempted style) and pleasing (especially in terms of temporal coherence) as possible. In order to tackle the problem of creating each of the different renderers, we were guided by the following list of analytic steps:

- Identify key-elements of the style (Definition of Style)
- Define a general problem statement (what problems have to be solved to implement the above style)
- Identify problems specific to: Real-time / Quality constraints, Hardware / Software limitations or Graphics API (Implementation-specific)
- Understand the standard approach taken by other authors
- Identify possible optimisations and extensions (to improve visual quality or further performance compared to the standard approach)

The main problems that had to be solved for each of the renderers, any novel concepts devised and applicable results are discussed next.

### 8.1 Comic Rendering

The model solution to comic rendering is so standardised that only a handful of variations exist, all of which produce the same visual output. Our challenge therefore lay in extending the boundaries of what is generally considered a solved problem.

#### 8.1.1 General Problems and Solutions

Comics or cartoons in general almost unanimously make use of the following stylistic devices:

- A thick dark outline of the silhouette
- Other thick dark lines detailing object characteristics
- Uniform or very banded shading

In order to implement a renderer replicating these devices, the following problems have to be solved:

- Identify the silhouette
- Identify other important folds and creases in the object
- Render these features with thick dark lines
- Shade the surface of the object in a single colour or
- Apply a heavily banded shading to the surface

The first three problems are addressed using the conventional solution. The silhouette and other object-detail is not identified on a geometric level in object-space, but on a pixel-level in screen-space. A simple two-pass algorithm can accomplish this. This approach is generally hardware-accelerated and thus extremely fast.

Implementing totally flat shading is not considered a problem, as it is catered for by the graphics API. Banded shading on the other hand needs to be simulated. The standard solution computes lighting information per vertex. More specifically, the diffuse light component is calculated. This normalised lighting value is then used to index into a one-dimensional shade-texture. A shade-texture contains the mapping of light-values to banded shading-value and can be seen as a quantisation-function. The reason to implement this function by means of texturing is that textures-vertices are automatically interpolated across surfaces, so that smooth shading boundaries can be achieved.

### 8.1.2 Novel Concepts

As stated in Section 8.1.1, the standard solution to comic rendering only takes into account a viewer-independent diffuse light component. Specular light interaction which is view-dependent and occurs on shiny objects like plastic, metal or polished surfaces is not taken into account, yet it increases the visual appeal in photorealistic contexts (e.g. raytracing) tremendously. We argue it can do the same in a non-photorealistic context.

While the lighting model for specular reflection is well-studied and understood, it has never been used in connection with computer generated comic rendering before (some real-life artists introduce similar lighting effects into their work, underlining the relevance of this contribution). We remedy this situation by designing and implementing a specular light-component that is closely modelled on a physical approximation but at the same time displays a very distinct but highly configurable cartoon quality.

Owing to the fact that the above-mentioned specular cartoon component presents a considerable computational overhead due to its view-dependence, we have to address this problem in the form of various performance enhancements. Our solution is to *approximate* the *view-vector* to be constant for the entire object, which allows us to re-write the formula of the specular component into a more efficient form. Unfortunately, under perspective projection the approximation is only correct for an infinite distance between the object and the viewer and worse the closer the two are. If the viewer is too close to the object, artefacts in the form of holes can appear in the object. We address this problem by applying a

*perspective correction*. This means that we estimate the maximum error of our view-vector approximation and draw more triangles than usually necessary to cover the front-side of the object to counteract. This solution works very well, because the cost of drawing additional triangles is lower than that of correct view-vector calculations per vertex.

To minimise the number of lighting calculations as well as triangles to be rendered, we sort faces into front and back-facing. This is to a limited extent also done in the standard approach, but we identified a new and fast way of establishing face orientation which also represents a flexible level-of-detail adjustment. The main concept that makes our idea work is Euler's relation [111], stating that for closed, convex polygons the number of faces is larger than the number of vertices (the actual relation is more specific than this). We therefore determine the orientation of vertices instead of triangles and then deduce the orientation of triangles from the orientation of their defining vertices. In general vertices do not have the same normals as the triangles they comprise, but for finely tessellated objects this approximation is fairly good. This in turn is important, because it means that our approximation is better the more detailed an object is, which usually implies that it takes longer to render so optimisations for this type of object are particularly valuable. In practice, the level-of-detail adjustment mentioned above determines how many vertices have to be front-facing in order for a triangle to be considered front-facing (either one, two or all). Demanding that all vertices be front-facing before a triangle is marked as front-facing is the most restrictive, resulting in the least amount of front-faces and the best performance. As we already mentioned, our approximation works very well for very detailed objects, so that a great performance increase can be achieved. The requirement of only one front-facing vertex per triangle is the least restrictive and therefore results in the best visual results and worst performance. For low-detail objects performance is quite good anyway, so that visual quality is of greater concern here. Since the viewer-object distance also plays into the determination of face-orientation, the number of vertices used can be made dependent on it, allowing to trade performance against quality if the object is in the distance and vice versa.

On top of the just-mentioned advantages, we can re-use the vertex-orientation calculations for the specular light computations, in essence obtaining the latter virtually for free. This fact, even though most convenient, does not restrict the applicability of either optimisation – both are valid and important in their own right and can be exploited independently.

Other novel ideas include the use of a coloured silhouette (instead of the standard black one) and comic-style base-textures to enhance the visual appeal of our renderer. Both of these extensions can easily be implemented without greatly affecting performance.

### 8.1.3 Results

We have shown that comic rendering for our test-objects can be performed at between 40-450 frames per second. For objects of more than 10,000 triangles our extended comic renderer, including the various optimisations and extensions discussed in Section 8.1.2, even outperforms the standard renderer which

only incorporates a diffuse light component. We have therefore demonstrated the effectiveness of our geometric approximations, which are in no way limited to comic rendering.

## 8.2 Sketch Rendering

Probably the largest variety of solutions to NPR rendering exist for sketch style rendering. Two main approaches exist, which see sketching either as an *outlining* style or a *hatching* style. We do not necessarily make this distinction and believe that both styles can easily complement each other.

### 8.2.1 General Problems and Solutions

While deliberately ignoring the conventional distinction of established sketching styles, we define a typical sketch as implementing the following concepts:

- Drawn by hand (Randomness / Uncertainty-Factor)
- *Economy of line* (little, but important object detail)
- Few colours used (monochrome)

Implementing these concepts requires solutions to the following set of problems:

- Creation a *manual-production* look
- Identification of important object detail (semi-) automatically
- Rendering the specific object-detail with:
  - Deliberate Imperfections
  - Rudimentary hints at shading if requested

Identification of important object-detail (especially the silhouette) or manual tagging of such are problems with well-established solutions. An edges can easily be considered belonging to the silhouette if one of its defining faces is front-facing while the other one is back-facing. Stochastic methods and optimised edge-traversals are readily available in the literature and can be applied where necessary. More interesting for us personally was the introduction of deliberate imperfections, which make computer-generated sketches look far more convincing and as if produced by hand. We follow two totally different paths, both of which produce convincing and efficient sketches. Firstly, we devise a random perturbation sketcher which actually modifies the underlying geometry of our objects and secondly, we use texturing to simulate random disturbances.

The concept of hatch-shading through perspective texturing is borrowed from published work, but we find new and more efficient ways of implementing it.

### 8.2.2 Novel Concepts

Firstly, we devise the ideas of *importance-functions* and *uncertainty-functions* to facilitate a generalised discussion of the topic of sketch rendering. Importance-functions define which edges of an object are considered for rendering purposes and we identify a small variety:

- Constant Importance-functions:
  - Edges above certain length
  - Edges with small dihedral angle
  - User-specified edges
- Variable Importance-functions:
  - Edges belonging to the silhouette
  - Edges belonging to other important object-detail

Uncertainty-functions simulate the desired manual production look and can be applied singularly or concatenated in order to produce a desired effect. Some uncertainty-function identified by us include:

- Repetition of Lines
- Segmentation of Lines
- Perturbation of Lines / Segments
- Applying Offsets to Lines / Offsets
- Any combination of the above

To implement these uncertainty functions, we first use a recursive sub-division algorithm, but the additional cost of recursive programming negatively effects performance. Instead we devise a new method we call *object-segmentation*. Each edge in an object is considered a separate object and a transformation-matrix is computed at load-time to transform a well-defined unit-vector into the edges position and orientation. We then replace the unit-vector with one of a selection of pre-perturbed (i.e. uncertainty-functions are computed and applied once-off at load-time instead of continuously at run-time) lines, which we store in optimised displaylists. This approach is highly efficient and even easier to implement than the recursive version.

Our alternative solution using texturing stores uncertainty information in the form of stroke-textures, which makes them highly customisable. Real pencil, chalk or coal-strokes can be scanned and converted to textures to maximise the natural media look we strive for. Even though similar solutions have been implied by some authors, we explicitly address the issue of temporal coherence associated with animation of such a solution. The problem is that the conventional silhouette condition is either met by an edge or not. The transition is instantaneous, resulting in edges popping in an out of the scene, which is visually distracting. Our solution gracefully fades edges in which are about to become visible and those out, which are about to disappear. All of this is done using an ingeniously simple but effective method: by multiplying the dot-product information necessary to establish the silhouette-condition we obtain a continuous measure of the silhouette condition. We know that for negative values of this product the edge-condition is met, so we design a function that increases as the multiplication product approaches positive zero, stays constant (full exposure) in the negative interval and is symmetric about  $\pi$ . With this approach, we do not need to be aware if an edge is about to appear or rather disappear, because the function will work either way. The simplicity of the solution entails that performance is basically not



affected, except that more edges will be rendered (those which would usually produce the edge-popping) which obviously results in a proportional slow-down.

Our hatch-shading sketcher uses projective texturing like other known implementations, but instead of using expensive surface-subdivisioning, we devise several very efficient texturing solutions. Our first suggested solution employs multi-texturing. Hatching simulates light-intensity via density of hatching strokes. This means that strokes of different densities have to be applied to different regions of the object and the problem is allowing shading regions to run smoothly over single surfaces. We achieve this by including the banded-shading approach of the comic rendering solution as the bottom-layer in a multi-texturing scheme. For different shading regions, multiple passes are necessary because multi-texturing capabilities are commonly limited to two texture units.

A still more innovative solution was found in the form of 3D texturing. While the concept of 3D textures logically follows as an extension of 2D texturing, it seems as if very few uses have so far been found for it. We remedy this fact by claiming that 3D texturing is an ideal candidate for hatch-shading. All the different hatch densities required can be stacked into a 3D texture in layers. Projective texturing is then applied as usual, but the lighting value indicating which hatch-density is to be used can directly index the third dimension of the 3D texture. This novel way of hatch rendering allows us to render any amount of hatch-densities, which smoothly blend into one another, in only a single rendering pass and without the need for costly surface sub-divisioning or multi-pass rendering.

### 8.2.3 Results

Various applicable results were found. Firstly, we showed conclusively that real-time sketch rendering can be performed anywhere between 20-275 frames per second, depending on the rendering style used and object complexity. Secondly, apart from demonstrating how natural-looking sketches can be produced with either of two methods (random perturbation or texturing) we successfully dealt with temporal issues, the solution to which can be applied to any situation where rendering of edges is required. Abstraction to other rendering situations is also imaginable.

## 8.3 Painterly Rendering

Only very little work is published on real-time painterly rendering and the existing work is split into to camps: Brushes that are fixed in *object-space* and brushes that are fixed in *screen-space*. Alternative solutions not using brushes at all usually suffer from extremely poor performance in addition to lacking in configurability. Once again our approach varies slightly from the established techniques and provides surprising and interesting visual results.

### 8.3.1 General Problems and Solutions

The sheer number of different painting styles makes a classification extremely difficult. We therefore resort to identifying key elements in the process of painting (and not limited to artistic painting):

<b>Produced by hand</b>	<ul style="list-style-type: none"> <li>• Deliberate spatial (and temporal) flaws</li> </ul>
<b>Brushes</b>	<ul style="list-style-type: none"> <li>• <i>Shape</i></li> <li>• <i>Size</i></li> </ul>
<b>Paints/Colours</b>	<ul style="list-style-type: none"> <li>• <i>Palette</i></li> <li>• <i>Consistency</i> (Watery, Oily, ...)</li> <li>• Transparent/Opaque</li> <li>• Blending/Mixing</li> </ul>
<b>Technique</b>	<ul style="list-style-type: none"> <li>• <i>Choice of {Brush, Colour, Strokes}</i></li> <li>• <i>Strokes</i> <ul style="list-style-type: none"> <li>• Direction, Length, Pressure</li> <li>• Number</li> <li>• Distribution</li> </ul> </li> </ul>

As with the sketching renderers, human imperfections and natural media reproduction play an important role in this renderer. We can omit a re-iteration of problems demanding to be solved in this case, because we need to methodically implement each of the above concepts of *brushes*, *paints* and *technique*.

Painterly filters are considered standard content of almost any image processing package. These filters are usually implemented using convolution filters and are thus computationally extremely expensive, which rules them out for use in real-time applications. In addition to performance constraints, configuration of these filters is commonly very limited and cumbersome. Reproduction or simulation of natural media is almost impossible and rarely achieved.

Another standard solution exists which uses textured quadrilaterals to simulate brushes. These are then either affixed to object geometry or screen co-ordinates. The former approach results in a much superior temporal quality compared to the latter approach, because brushes move with objects. Unfortunately, the number of brushes necessary to fill the screen are significantly higher and brush-management becomes increasingly expensive with increased object-count or complexity. The alternative approach of fixing brushes in screen-space is easy and efficient, but produces what is called the *shower-door-effect*, which is very unconvincing under animation. Other approaches like total random brush positioning also tend to suffer from the same effect in addition to producing a large amount of visual noise.

As brush placement is usually the main concern of work on real-time painting, such issues as *paint* and *technique* are often neglected. Papers dealing with these issues on the other hand are generally more concerned with physically correct models as opposed to real-time performance. We do address these issues and point out simple but effective solutions along with examples of most of our suggestions.

*Reference images*, which are used to select suitable colours for brushes and can aide in brush placement and selection are rendered in real-time in our system (i.e. not only is the reference image repainted in real-time, but the performance results include the manufacture of the reference image itself). In order to

minimise expensive screen-capture operations and realise co-existence with other renderers or scene elements, we discuss a variety of possible solutions, including

- Auxiliary and Stereo Buffers
- Stencil Buffer
- Pixel Buffers (Pbuffers)

### 8.3.2 Novel Concepts

Even though textured brushes are the only real-time solution to painterly rendering to our knowledge, we were able to extend the established implementation of a simple brush shape to that of elaborate brush strokes, including properties such as:

- Speed and Motion
- Stroke shape and length
- Paint capacity, opacity and emission
- Paint texture (such as bristle lines or air bubbles one encounters on dried paint)

These extended brushes are neither fixed in object- nor screen-space in our renderer. Instead we experiment with semi-random motion (only semi, because we also want to avoid the shower-door-effect). We produce a solution that moves brushes in a pseudo-random Brownian motion in relation to object position. This minimises the shower-door-effect while generating surprisingly soothing visuals.

In order to further minimise the amount of brushes necessary to completely fill the screen-extent of the rendered object, we maximise brushes that are located over object-pixels, without fixating them. We achieve this through a technique similar to *blue-screening* or *matting*, which can easily be incorporated in two of the reference-acquisition techniques. Brushes that are not located over object-pixels are said to be *invisible*, because they will not be rendered (lest they upset other scene elements). Establishing visibility of brushes is done via look-up in the screen-capture buffer and comparison with the matte-colour. Brushes know when they are invisible and will try to find their way back into visible regions as soon as possible. We do this via motion-inversion. If brushes fail to become visible within a pre-defined *time-to-live*, they are re-initialised at random. Knowing the number of visible brushes at any given time allows us to adjust their screen-size to always guarantee a given object fill-ratio.

### 8.3.3 Results

In a static situation our output images can compete with any commercially available convolution filter approaches. In an animated context our solution presents itself to be pleasingly different from existing ones in that brushes move smoothly and freely over objects as if guided by copious invisible hands while not being limited to fixed positions on the geometry of these objects. Still, brushes will move in unison with the general movement of objects so as to minimise any possible shower-door-effect.

As brush count has more impact on performance than object detail, our solution is predictably independent of object complexity (within reasonable limits). In excess of 10,000 brushes can be rendered at more than 10 frames per second, but our personal preference of between 2-3,000 brushes (in terms of output resolution achieved) renders well within 25-30 frame per second.

## 8.4 Super-realistic Rendering

The inclusion of realistic renderers or even photo-realistic rendering within a thesis dealing with real-time NPR probably needs justification. Our reasoning is that common photorealistic techniques are not as yet real-time (given very recent and few exceptions) and real-time techniques are yet some ways from being photo-realistic. A renderer which achieved photo-realistic quality in real-time is therefore more than photo-realistic (we call it *super-realistic*) and therefore non-photorealistic. We also see a trend in the somewhat neglected world of NPR to only include a very limited set of rendering styles (i.e. the ones previously discussed) and want to challenge a broader view of what can and should be considered NPR. In this light, if standard NPR is on one side of normal real-time photo-realism (i.e. that of games) then super-realistic rendering is on the exact opposite side.

### 8.4.1 General Problems and Solutions

The very fact that only about a handful of proper photorealistic rendering systems exist (and that mainly through advances in chip-technology as opposed to break-throughs in rendering algorithms) is an indication of the complexities involved in such an undertaking. We calmly acknowledge this situation and attempt a radically different approach, which is gaining popularity of late. Image based rendering in broad terms means modifications of existing images to produce new ones. This may seem rather trivial, but it implies that each frame of an animation or an interactive scene does not have to be re-generated from scratch, but that the whole, or parts of, a scene can be cached and modified in order to complete the rendering. Texture-mapping is an example of this: Instead of specifying a ridiculous amount of geometrical data along with vertex-colour information, we can simply apply a pre-generated texture containing all the necessary information.

Our general solution to the problem (equal to one developed at roughly the same time, but independently) is to display pre-generated images of objects instead of creating these objects at run-time. Since we have total control over the input-images, we can define their quality at will. The obvious problem is that in an interactive situation we will want to translate, scale and rotate a given object. As images are easily scaled and translated, we need to address the issue of rotation. A general rotation of an object using only a single object is impossible if the object is not perfectly spherically symmetrical. In the other extreme we would need one image for every possible orientation of the object, another impossibility. One way out of the problem is to use uniformly distributed (on the surface of a sphere surrounding the object) samples and choose the nearest sample to a given orientation. This is what we call the *standard approach*, because we found one system that uses it (no other system known to us at the writing of this thesis attempt a similar renderer). We on the other hand go beyond the nearest sample approach and implement a novel but yet surprisingly familiar solution.

Our advanced solution relies on a multi-pass interpolation scheme. Related issues such as triangle-overlapping and triangle-flipping are addressed and efficiently solved using one of two possible depth-buffer solutions (multi-clear or layered rendering).

#### 8.4.2 Novel Concepts

By interpolating between neighbouring sample images A and B, we can simulate and synthesise all the missing samples between A and B. The information necessary to perform such an interpolation are a few but strategically well-placed control-points and paths along which these control-points move from their origin to their destination. The origin would be some object feature in image A, while the destination would be the same feature in image B (but problems may arise when this feature is missing in image B. In many cases it can be *carved out* of other image areas.). We achieve this by triangulating the complete set of control-points and texturing these triangles with fixed parts of image A. As the control-points move along their paths, image A is smoothly transformed into image B. We thus accomplish linear interpolation of image-pairs. If we also apply the reverse process (move control-points from destination to origin, with the same triangulation, but different texture co-ordinates) we get bi-linear interpolation, which greatly improves visual quality without any additional data required. The effect thus realised is called *tweening*, and usually considered a fairly complex task. With the same solution but by using neighbouring images of different content, we can easily create *morphs* in real-time, which would normally take several minutes to produce. As mentioned above our solution is therefore not technically new, but its similarity with existing techniques is accidental and its implementation seems to be rather novel (we could find no other morphing or tweening system using our highly optimised and mostly hardware-accelerated method).

Acquisition and identification of control-vertices is an interesting and demanding topic in itself and our system only approximates a complete solution. Nonetheless, we defined a set of rules which can be used in a short-circuit evaluation scheme to place suitable control-points. We name pixel areas where appropriate control-vertices may be situated *Landmarks*. Our set of rules reads as follows:

1. Landmarks may not lie too close to image boundaries (discontinuities)
2. Landmarks may not lie too close to one another (redundancy)
3. Landmarks must lie on regions of high frequency (it is implied that most of the landmarks are enclosed by edges)
4. Normalised Cross Correlation (NCC) value of surrounding Landmarks varies significantly from Landmark under inspection (this makes a correct match in the adjacent Image more likely)
5. Not more than a certain number of NCC matches may be found (uniqueness)
6. { Landmarks can be found in the original and neighbouring image (self-verification) }

While some of these rules still need revising and tweaking of input parameters to these rules is difficult, we obtain promising results onto which further work in this direction can be based.



With the availability of a real-time super-realistic renderer we either implemented or propose a multitude of novel application areas, which include:

- Extreme Low Bandwidth video-conferencing
- Facial Character Animation
- Restoration of Film and Video material

#### **8.4.3 Results**

We successfully demonstrated the superior visual quality of interpolated images compared to non-interpolated samples. What's more, we were able to implement a demonstration system, which can perform high-quality bi-linear interpolation in excess of well over 400 frames per second. Interpolation of these results demonstrate that even a more advanced system using further rendering passes can still perform well within real-time constraints.

### **8.5 Extensions and Future Work**

In this Section we briefly expand on the possible, likely and theoretical extensions to our work as it stands.

#### **8.5.1 Standard Improvements**

As with most implementations, we can always try to find faster or better ways of accomplishing certain tasks. In order to be generally applicable most of our discussions did not include advanced implementation-specific optimisations, such as assembler-level coding of critical and computationally expensive algorithmic Sections. For example it seems common practice in the field of real-time photorealistic rendering to include hardware-optimisations like Pentium III streaming instructions as relevant work.

Another very real and promising development with respect to this is that of modern graphics card technology, which are becoming every more customisable at ever lower levels. The concept of programmable vertex shaders for example ([52], [39]) enables us already today to fully implement comic shading in hardware. As the demand for these GPU (graphics processing unit) programming features is high and rising, we expect the functionality of future graphics card to be less restricted to photorealistic approximation and more readily available for NPR implementations.

#### **8.5.2 Expansion of Definition**

We have already tried to stretch the common conception of NPR by including a super-realistic renderer and mentioning SIRDs (Stereograms). While the established NPR styles seem very much standardised, we hope for an expansion of this implicit definition in order to completely exploit the full potential held by NPR. We should try to develop new artistic styles which are not mere reproductions of known human artwork, but are possibly only realisable in a computing environment (like to the animation of paintings).

If we consider the term *rendering*, as it is used within this thesis, to mean the translation of abstract (geometrical) information into a concrete physical representation on the screen of a display-device, then we could also expand the concept of NPR renderers to non-visual ones. While risking to sound philosophical, we argue that anyone who has ever dreamed would agree that an imagined scene can look more realistic or much less realistic than anything they have ever experienced in real life. With this in mind, we can imagine physiological, psychological or acoustic renderers to be invented and used. For example, we could imagine a situation in which the user of an NPR system lies on a comfortable couch, eyes closed and listens to a scene being rendered: “you are in a small room with a large panoramic window on the west side...”. Such renderers have the potential of producing, through relatively simple means, a much larger sensory stimulation than those restricted to the visual sense.

## 8.6 Contribution of Thesis

In our problem-statement in Section 1.1, we stated the following set of goals for this thesis:

- 1) Enhance the *performance* of existing NPR renderers
- 2) Enhance the *visual quality* of existing NPR renderers, while remaining within real-time constraints
- 3) Create new NPR renderers that perform in real-time

We furthermore defined what we consider to be a valuable and relevant enhancement with respect to any NPR style:

- a) Increase its creative or expressive potential
- b) Increase its communicative potential
- c) Increase its rendering speed
- d) Expand its definition (i.e. add visual qualities which have not yet been considered)
- e) Employ new techniques or use existing techniques in new ways

We will now conclude this thesis by confirming that all our original goals have been achieved.

We expanded the definition of the standard comic renderer by adding an additional lighting component (2-d). We also added various optimisations that enable this enhanced version to render at almost the same speed as the original version (1-c).

We invented new ways of rendering random perturbation sketches using object segmentation and the use of uncertainty functions stored in displaylists (1-e). While providing the same visual output as the standard solution, our new technique is considerably faster (1-c). We extended the visual quality of standard coal sketch renderer by designing an edge-fading mechanism that allows for smooth fading of edges under animation (1-d). Hatch rendering was revolutionised by using 3D texturing, which was never employed for this purpose before (1-e). As support for 3D texturing is limited we ingeniously recycled the

standard comic rendering mechanism (1-e) to produce continuous and smooth hatching over arbitrary triangulated surfaces, without the need for surface subdivision or multi-pass rendering.

We increased the creative and expressive potential of painterly rendering, but adding new properties to standard textured brushes, allowing them to be used more creatively and with greater flexibility (2-a). We also expanded the standard conception for positioning of brushes by allowing them to move pseudo-randomly but relative to the object, thus creating a new and interesting visual appearance (2-d).

Finally, we developed a completely new NPR renderer, which not only expands the definition of NPR itself, but which also uses simple and readily available texturing and interpolation techniques in order to achieve real-time performance (3, 1-e). The communicative potential of this renderer is considerable, due to its many interactive application areas, a great many of which we identify and describe (2-b).

We have made real-time enhancements to every standard NPR style and even developed a new one. It can therefore be stated with conviction that every single goal of this thesis has been successfully realised.

## 9 References

### 9.1 Literary References

- [1] Adams Scott, "*Dilbert*", Comic Strip and Television Series, Syndicated through United Media.
- [2] Appel A., "*The notion of quantitative invisibility and the machine rendering of solids*", Proceedings of ACM National Conference, pp. 387–393, 1967.
- [3] Apple Computers Inc., "*Quicktime VR Authoring*", <http://www.apple.com/quicktime/qtvr/>
- [4] Baxter B., Scheib V., Lin M. C., Manocha D., "*DAB: Interactive Painting with 3D Virtual Brushes*", Proceedings SIGGRAPH, 2001.
- [5] Bertram Anthony, "*1000 Years of Drawing*", Studio Vista Limited, London, 1966.
- [6] Bier E.A., Sloan K.R., "*Two-part texture mapping*", IEEE Computer Graphics and Applications, 6(9), 40-53, September 1986.
- [7] Blinn J.F., Newell M.E., "*Texture and reflection in computer generated images*", Comm. ACM, 19 (10), 362-7, 1976.
- [8] Blinn, J.F., "*Simulation of wrinkled surfaces*", Computer Graphics, 12(3), 286-92, Proceedings SIGGRAPH 1978.
- [9] Cohen J.M., Hughes J.F. and Zeleznik R.C., "*Harold: A World Made of Drawings*", Proceedings of NPAR, 2000.
- [10] Cohen Jonathan, "*Systems for sketching in 3D*", Undergraduate Thesis, Brown University, Providence, RI, May 2000.
- [11] Cohen M.F., Greenberg D.P., "*A radiosity solution for complex environments*", Computer Graphics, 19(3), 31-40, July 1985.
- [12] Cohen M.F., Greenberg D.P., Immel D.S., "*An efficient radiosity approach for realistic image synthesis*", IEEE Computer Graphics and Applications, 6(2), 26-35, March 1986.
- [13] Columbia Pictures, "*Final Fantasy*", Feature Film; by Columbia Pictures and Square Pictures; © 2001 Final Fantasy Film Partners.
- [14] Corrêa W. T., Jensen R. J., Thayer C. E., Finkelstein A., "*Texture Mapping for Cel Animation*", Proceedings SIGGRAPH, July 1998.
- [15] Curtis C. J., Anderson S. E., Seims J. E., Fleischer K. W., Salesin D. H., "*Computer-Generated Watercolor*", Proceedings SIGGRAPH, 1997.
- [16] Debevec Paul E. et al, "*Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach*", Proceedings SIGGRAPH, pp. 11-20, 1996.
- [17] Deussen O., Hiller S., van Overveld C. and Strothotte T., "*Floating Points: A Method for Computing Stipple Drawings*", Proceedings Eurographics, 19(3), 2000.
- [18] Deussen O., Strothotte T., "*Computer-Generated Pen-and-Ink Illustration of Trees*", Proceedings SIGGRAPH, pp. 13–18, 2000.
- [19] FOX Television, "*Futurama*", Television Series, FOX Corporation, [www.fox.com/futurama/index.html](http://www.fox.com/futurama/index.html)

- [20] Gamasutra, "Ups and Downs' of Bump Mapping with DirectX 6", [www.gamasutra.com / features / 19990604 / bump\\_01.htm](http://www.gamasutra.com/features/19990604/bump_01.htm)
- [21] Gardener G.Y., "Visual simulation of clouds", *Computer Graphics*, 19 (3), 297-303, 1985.
- [22] Geisel Theodor (Dr. Seuss), "The Lorax", Random House, New York, 1971.
- [23] Girshik A., Interrante V., "Real-time Principal Direction Line Drawings of Arbitrary 3D Surfaces", *Proceedings SIGGRAPH*, pp. 271--271, 1999.
- [24] Goldstein Nathan, "The Art of Responsive Drawing", Prentice Hall, Inc. N.J., 1977.
- [25] Gooch A., Gooch B., "Using Non-Photorealistic Rendering to Communicate Shape", Course Notes for SIGGRAPH Conference, 1999.
- [26] Gooch A., Gooch B., Shirley P., Cohen E., "A Non-Photorealistic Lighting Model For Automatic Technical Illustration", *Proceedings SIGGRAPH*, pp. 447--452, July 1998.
- [27] Goral C., Torrance K.E., Greenberg D.P., Battaile B., "Modelling the interaction of light between diffuse surfaces", *Computer Graphics*, 18(3), pp. 212--22, July 1984.
- [28] Hall Peter, "Comic-strip rendering", Technical Report CS-TR-95/2, Victoria University of Wellington, 1995.
- [29] Hall R.A., Greenberg D.P., "A testbed for realistic image synthesis", *IEEE Computer Graphics and Applications*, 3(8), pp. 10--19, November 1983.
- [30] Hawkins Joyce M., "The Oxford Paperback Dictionary", Oxford University Press, Third Edition, 1988.
- [31] Heckbert P.S., "Survey of texture mapping", *IEEE Computer graphics and Applications*, 6 (11), pp. 56--67, November 1986.
- [32] Hertzmann Aaron, "Painterly Rendering with Curved Brush Strokes of Multiple Sizes", *Proceedings SIGGRAPH*, pp. 453--460, 1998.
- [33] Hertzmann Aaron, Perlin Ken, "Painterly Rendering for Video Animation", *Proceedings NPAR*, 2000.
- [34] Hill Francis S., Jr., "Computer Graphics", Macmillan Publishing Company, 1990.
- [35] Hoffman Donald D., "Visual Intelligence - how we create what we see", W\*W\*Norton ISBN: 0-393-04669-9, February 2000.
- [36] Holme Charles, "Pen, Pencil and Chalk - a Series of Drawings by contemporary European Artists", The Studio Ltd., London, 1911.
- [37] Immel D.S, Cohen M.F., Greenberg D.P., "A radiosity method for non-diffusive environments", *Computer Graphics*, 20 (4), 133-42, 1986.
- [38] Jones Michael, "The Flick in Flicker: Persistence of Vision and the Phi Phenomenon", [www.flicker.org/flickerinflicker.htm](http://www.flicker.org/flickerinflicker.htm), 1999.
- [39] Kilgard Mark J., "NV\_vertex\_program", Proposed Specification Document, © NVIDIA Corporation, September 2000.
- [40] Klein A. W., Li W., Kazhdan M.M., Corrêa W. T., Finkelstein A., Funkhouser T. A., "Non-Photorealistic Virtual Environments", *Proceedings SIGGRAPH*, pp. 527--534, July 2000.
- [41] Kowalski M.A., Markosian L., Northrup J.D., Bourdev L., Barzel R., Holden L.S., Hughes J., "Art-Based Rendering of Fur, Grass and Trees", *Proceedings SIGGRAPH*, pp. 433--438, 1999.



- [42] Lake et al, "Stylised Rendering Techniques For Scalable Real-Time 3D Animation", Proceedings NPAR, 2000.
- [43] Lander Jeff {jeffl@darwin3d.com}, "Game Developer Magazine Companion Source Page", [www.darwin3d.com/gamedev.htm](http://www.darwin3d.com/gamedev.htm).
- [44] Lansdown J. and Schofield S., "Expressive rendering: A review of nonphotorealistic techniques", IEEE Computer Graphics and Applications, May 1995.
- [45] Lengyel J., Praun E., Finkelstein A., Hoppe H., "Real-Time Fur over arbitrary Surfaces", Proceedings of Interactive Symposium on 3D Graphics, pp. 227--232, 2001.
- [46] Luebke D., "View-Dependent Particles for Interactive Non-Photorealistic Rendering", Proceedings of Graphics Interface, 2001.
- [47] Markosian L., Kowalski M., Trychin S., Hughes J., "Real-Time Nonphotorealistic Rendering", Proceedings SIGGRAPH, pp. 415--420, August 1997.
- [48] Markosian L., Meier B., Northrup J.D., Kowalski M.A., Holden L.S., Hughes J.F., "Art-based Rendering with Continuous Levels of Detail", Proceedings NPAR, 2000.
- [49] McCloud Scott, "Understanding Comics - The invisible Art", Harper Perennial, 1993.
- [50] McReynolds Tom, Blythe David, "Advanced Graphics Programming Techniques Using OpenGL", [www.sgi.com/Technology/OpenGL/advanced\\_sig98.html](http://www.sgi.com/Technology/OpenGL/advanced_sig98.html)
- [51] Meier Barbara J., "Painterly Rendering for Animation", Proceedings SIGGRAPH, pp. 477--484, August 1996.
- [52] Microsoft Corporation, "Cartoon Rendering in DirectX 8 Using Vertex Shaders", <http://msdn.microsoft.com/library/techart/DXVertex.htm>
- [53] Microsoft, "Win32 Developer's References - OpenGL Programmer's Reference", Microsoft Cooperation, © 1995. (Portions extracted from "OpenGL Programming Guide and the OpenGL Reference Manual", © Silicon Graphics, Inc.), 1995.
- [54] Microsoft MSDN Collection, "Architectural Overview for Direct3D", [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dx8\\_c/hh/dx8\\_c/graphics\\_understand\\_6ylv.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dx8_c/hh/dx8_c/graphics_understand_6ylv.asp)
- [55] Mitchell D.P., "Generating antialiased images at low sampling densities", Computer Graphics, 21(4), pp. 65--72, 1987.
- [56] Moore Gavin, "Talking Heads: Facial Animation in The Getaway", [www.gamasutra.com/features/20010418/moore\\_01.htm](http://www.gamasutra.com/features/20010418/moore_01.htm)
- [57] Namayu Kim et al, "Photo-realistic 3D virtual environments using Multi-view Video", Proceedings SPIE-EI-VCIP, 2001.
- [58] Nishita T., Nakamae E., "Continuous tone representation of three-dimensional objects taking account of shadows and interreflection", Computer Graphics, 19(3), pp. 23--30, July 1985.
- [59] Northrup J. D., Markosian L., "Artistic Silhouettes: A Hybrid Approach", Proceedings of NPAR, pp. 31--38, June 2000.
- [60] NVidia Corporation, "Technical Brief - Transform and Lighting (PDF)", [www.nvidia.com](http://www.nvidia.com)
- [61] NVidia, "NVIDIA OpenGL Extension Specifications", NVIDIA Corporation, March 2001.
- [62] Oi Jun, "An Easy Way to Make a STEREOGRAM", [www.tcp-ip.or.jp/~junoi/stereograms/theory/make.html](http://www.tcp-ip.or.jp/~junoi/stereograms/theory/make.html)

- [63] Opalach Agata, Maddock Steve, "Disney Effects Using Implicit Surfaces", Department of Computer Science, The University of Sheffield, September 1994.
- [64] OpenGL.org, "Orthogonal Illumination Maps", [www.opengl.org / News / Special / oim / Orth.html](http://www.opengl.org/News/Special/oim/Orth.html)
- [65] Panagou S. and Bangay S., "An Investigation into the feasibility of Human Facial Modeling", Proceedings SATNAC, 1998.
- [66] Perron Carin, "The Basics of Animation", [www.writer2001.com/animprin.htm](http://www.writer2001.com/animprin.htm), 2001.
- [67] Petrovic L., Fujito B. T., Williams L., Finkelstein A., "Shadows for Cel Animation", Proceedings SIGGRAPH, pp. 511--516, July 2000.
- [68] Praun E., Hoppe H., Webb M., Finkelstein A., "Real-Time Hatching", Proceedings SIGGRAPH, August 2001.
- [69] Raskar Ramesh, "Hardware Support for Non-photorealistic Rendering", SIGGRAPH Graphics Hardware, August 2001.
- [70] Raskar Ramesh, Michael Cohen, "Image Precision Silhouette Edges", Symposium on Interactive 3D Graphics, April 1999.
- [71] Reitberger Reinhold, Fuchs Wolfgang, "Comics - Anatomy of a mass Medium", Little, Brown and Company Boston Toronto, 1972.
- [72] Robinson Jerry, "The Comics - An illustrated History of Comic Strip Art", G.P. Putnam's Sons, New York, 1974.
- [73] Rössl C., Kobbelt L., "Line-Art Rendering of 3D-Models", Proceedings IEEE Computer Society, 2000.
- [74] Rössl C., Kobbelt L., Seidel H.P., "Line Art Rendering of Triangulated Surfaces using Discrete Lines of Curvature", Proceedings WSCG '2000, pp. 168--175, 2000.
- [75] Sabin Roger, "Comic, Comix & Graphic Novels - A History of Comic Art", Phaidon Press Ltd., 1996.
- [76] Sabin Roger, quoted from "Adult Comics – An Introduction", (The Pentagon Press Office was not able to give an exact reference for the research, but said it was undertaken by the US Army in around 1981), Routledge publishing, 1993.
- [77] Saito T., Takahashi T., "Comprehensible Rendering of 3-D Shapes", Proceedings SIGGRAPH, 1990.
- [78] Salisbury M. P., Wong M. T., Hughes J. F., Salesin D. H., "Orientable Textures for Image-Based Pen-and-Ink Illustration", Proceedings SIGGRAPH, pp. 401-406, 1997.
- [79] Samek M., Slean C. and Weghorst H., "Texture Mapping and distortion in digital graphics", Visual Computer, 2{5}, 313-20, 1986.
- [80] Schlechtweg S., Strothotte T., "Rendering Line-Drawing with Limited Resources", Proceedings of GRAPHICON'96, pp. 131--137, 1996.
- [81] Schödl A., Szeliski R., Salesin D. H., Essa I., "Video Textures", Proceedings SIGGRAPH, July 2000.
- [82] SciTech Software, Inc., "GLDirect (Direct3D wrapper for OpenGL)", [www.scitechsoft.com/](http://www.scitechsoft.com/)
- [83] Segal Mark, Akeley Kurt, "The OpenGL® Graphics System: A Specification (Version 1.2.1)", Silicon Graphics, April 1999.

- [84] Seitz S. M. and Dyer C. R., "View Morphing", Proceedings SIGGRAPH, pp. 21--30, 1996.
- [85] Siegel R., Howell J.R., "Thermal Radiation Heat Transfer", Washington DC: Hemisphere Publishing, 1992.
- [86] Sim Dietrich, "Cartoon Rendering and Advanced Texture Features of the GeForce 256 Texture Matrix, Projective Textures, Cube Maps, Texture Coordinate Generation and DOTPRODUCT3 Texture Blending", NVIDIA Corporation, 2000.
- [87] Simpson Ian, "Practical Art School - Twelve lessons in Drawing, Painting & Sketching", Tiger Books International, London, 1995.
- [88] Sousa M.C., Buchanan J. W., "Computer-Generated Graphite Pencil Rendering of 3D Polygonal Models", Proceedings Eurographics, 18(3), pp. 195--208, 1999.
- [89] Sousa M.C., Buchanan J. W., "Computer-Generated Pencil Drawing", Proceedings SKIGRAPH, 1999.
- [90] Sousa M.C., Buchanan J. W., "The edge buffer: A data structure for easy silhouette rendering", Proceedings NPAR, pp. 39--42, 2000.
- [91] Stewart David B., "Real Time", Embedded Systems Programming, [www.embedded.com/story/OEG20011016S0120](http://www.embedded.com/story/OEG20011016S0120), January 2001.
- [92] Streit L., Buchanan J., "Importance driven halftoning", Proceeding Eurographics, 17(3), pp. 207--217, 1998.
- [93] Streit L., Veryovka O., Buchanan J., "Non-Photorealistic Rendering using an Adaptive Halftoning Technique", Proceedings SKIGRAPH, 1999.
- [94] Stuttard D. et al, "A Radiosity system for Real Time Photo-Realism", Computer Graphics: Developments in Virtual Environments, pp. 71--81, June 1995.
- [95] ten Hagen P., Noot H., Ruttkay Z., "CharToon: a system to animate 2D cartoon faces", Proceedings Eurographics, 1999.
- [96] Thimbleby H. W., Inglis S., Witten I. H., "Displaying 3D Images: Algorithms for Single Image Random-Dot Stereograms", IEEE Journal Computer, 27(10), pp. 38--48, Oktober 1994.
- [97] Tonnhofer T., Gröller E., "Autostereograms - Classification and Experimental Investigations", 12<sup>th</sup> Spring Conference on Computer Graphics, 1996.
- [98] Twentieth Century Fox, "Titan A.E.", Feature Film; Twentieth Century Fox Home Entertainment; [www.titanae.com/index\\_frames.html](http://www.titanae.com/index_frames.html)
- [99] Van Wallendael, "Gestalt Psychology I", University of North Carolina at Charlotte, [www.uncc.edu/Ivanwall/history/lec25-26.html](http://www.uncc.edu/Ivanwall/history/lec25-26.html), 2000.
- [100] Viacom Incorporated, "Rugrats - the Movie", Feature Film; © Nickelodeon, <sup>TM</sup> Viacom Inc., [www.nick.com/all\\_nick/movies/rugrats\\_paris/index.jhtml](http://www.nick.com/all_nick/movies/rugrats_paris/index.jhtml)
- [101] Wald I., Slusallek P., Benthin C., Wagner M., "Interactive Rendering With Coherent Raytracing", pp 153-164, 20(3), Proceedings of Eurographics 2001.
- [102] Wallace J.R., Cohen M.F., Greenberg D.P., "A two-pass solution to the rendering equation: a synthesis of ray-tracing and radiosity methods", Computer Graphics, 21 (4), 311-20, July 1987.
- [103] Watt A., Watt M., "Advanced Animation and Rendering Techniques", Addison Wesley, ACM Press, 1992.
- [104] Webpage, "Virtual Reality for the rest of us", <http://www.letmedoit.com/qtvr/index.html>

- [105] Webpage, BroadCast 2000, Real-time video-editing Software, <http://heroines.sourceforge.net/bcast2000.php3>.
- [106] Webpage, "Delaunay triangulation", <http://cage.rug.ac.be/~dc/alhtml/Delaunay.html>
- [107] Webpage, "Geometry in Action Delaunay Triangulation", [www1.ics.uci.edu/~eppstein/gina/delaunay.html](http://www1.ics.uci.edu/~eppstein/gina/delaunay.html)
- [108] Webpage, "Keith's Stereogram FAQ", [keith@rhythm.com](mailto:keith@rhythm.com), [www.rhythm.com/~keith/autoStereoGrams/stereogramFaq.html](http://www.rhythm.com/~keith/autoStereoGrams/stereogramFaq.html)
- [109] Webpage, "MPEG4 Standards Definition", <http://mpeg.telecomitalia.com/standards/mpeg-4/mpeg-4.htm>
- [110] Webpage, Olympus Digital, "360° View of Camera C-2040Z", [www.olympus-innovation.com/C-2040/c2040qtrv.html](http://www.olympus-innovation.com/C-2040/c2040qtrv.html)
- [111] Webpage, The Geometry Junkyard, "Seventeen Proofs of Euler's Formula:  $V-E+F=2$ ", [www1.ics.uci.edu/~eppstein/junkyard/euler/](http://www1.ics.uci.edu/~eppstein/junkyard/euler/)
- [112] Webpage, "Virtual Reality Special Interest Group", Rhodes University, [www.cs.ru.ac.za/vrsig/#RhoVer](http://www.cs.ru.ac.za/vrsig/#RhoVer)
- [113] Webpage, "Voronoi-Delaunay Applet", [www.cs.cornell.edu/Info/People/chew/Delaunay.html](http://www.cs.cornell.edu/Info/People/chew/Delaunay.html)
- [114] Winnemöller H., Bangay S., "Super-realistic Rendering using Real-time Tweening", Proceedings SATNAC, 2001.

## 9.2 List of Animations

In order to show various aspects of our NPR renderers and system, we provide a variety of animations that focus on these aspects. Many of these animations include a voice-track explaining the individual issues addressed. For animations without sound, we briefly explain their significance in the following listing.

- Animation A : Hidden Line Removal and Custom Background, "background&HLR\_w\_sound.avi"
- Animation B : Standard Comic style rendering and Extended Comic rendering, "comic\_w\_sound.avi"
- Animation C : Specular Highlight Approximation with Chromemap, "specular\_comic\_w\_sound.avi"
- Animation D : RPS and Coal Sketchers, "sketching\_w\_sound.avi"
- Animation E : Hatching style sketch renderer, "Hatching\_w\_sound.avi"
- Animation F : Hatching using a 3D-Texture, "3D\_Texture\_hatching.avi" (Light source rotating over angled plane)
- Animation G : Painterly Rendering with smart Brushes, "Painterly\_w\_sound.avi"
- Animation H : Super-realistic rendering, "SR\_Rendering\_w\_sound.avi"
- Animation I : Tweening of Teapot over 15 degrees, "Teapot\_Tweening.avi"
- Animation J : Animation of Character, "tween1\_cnn.avi"
- Animation K : Comparison of Source images vs. Tweened source images, "Bear\_side\_by\_side.avi" (LHS: Nearest sample approach; RHS: Our Super-realistic rendering approach. Pink dots mark original samples, all other frames are interpolated)
- Animation L : Expression synthesis by image deformation, "expression\_tweening\_sad.avi" (LHS: bi-linear interpolation of two input images; RHS: linear interpolation of single input image)
- Animation M : Morphing using our Super-realistic renderer, "morph1\_faces.avi"
- Animation N : Tweening for Video restoration, "video\_restoration.avi" (LHS: Only first and last frame are original, other frames tweened. RHS: original video material, quality and temporal resolution)



### 9.3 Table of Figures

Figure 1 - What you see is not what you get: "The Treachery of Images" (1929), René Magritte (1898-1967), Los Angeles County Museum of Art, Los Angeles, CA. ....	3
Figure 2 - Historical Comics (taken from [72]): a) Palaeolithic painting in the Altamira Cave, Spain; b) A Section of the Bayeux Tapestry (William's Army attacks the castle of Divan).....	6
Figure 3 - Abstraction as a tool for self-identification [49].....	6
Figure 4 - Two extreme Examples of Sketching: a) "Woman on Her Death Bed" ([24]); b) "Two Lionesses" ([5]).....	8
Figure 5 - Vectors: a) definitions; b) Dot product .....	21
Figure 6 - Edge issues: a) Correct computation; b) Approximation; c) Artefacts .....	22
Figure 7 - Custom Background Clear: a) tileable wrinkled paper texture; b) sketched camera on custom background.....	25
Figure 8 - HLR: a) standard approach; b) if background is present; c) with colour-masking .....	27
Figure 9 - Data Ratios for Test Objects .....	30
Figure 10 - Default Renderer Examples: a) Deer5; b) Barney .....	32
Figure 11 - Various Comic Styles [75].....	35
Figure 12 - Typical Comic Style: A scene from Futurama [19].....	36
Figure 13 - Comic two-pass rendering principle .....	39
Figure 14 - Standard Comic rendering: a) Rocket; b) Dog .....	40
Figure 15 - Face sorting without perspective correction at different distances to the object.....	43
Figure 16 - View rays at (a) finite and (b) infinite distance from the object .....	44
Figure 17 - Explanation for Object holes.....	44
Figure 18 - Angles in an extreme case.....	45
Figure 19 - Determining Face orientation: a) Standard; b) with correction offset.....	45
Figure 20 - Silhouette: a) Without perspective Correction; b) With perspective Correction .....	46
Figure 21 - Border Region Faces vs. normalised Distance .....	46
Figure 22 - a) 1D shade texture; b) 2D shade texture; c) example map.....	51
Figure 23 - a) Standard OpenGL shading; b) standard comic style; c) extended comic style .....	52
Figure 24 - Extended Light Model: a) Diffuse Component; b) Shade-Map.....	53
Figure 25 - Specular Component: a) Exact; b) Approximate; c)Difference.....	54
Figure 26 - A Collection of Shade-maps (a-e) .....	55
Figure 27 - Statue of Liberty: a) Monochrome; b) Highly reflective Metal shading .....	56
Figure 28 - More examples: a) Tail of Slinky the Dog; b) A Blender .....	56
Figure 29 - Silhouette Colour: a) Varied; b) Single.....	57
Figure 30 - Comic style with multi-texturing: a) Spiderman; b) Tigger; c) Forklift.....	58
Figure 31 - Some sketch examples: a) Outline to be coloured in {Charcoal}; b) Composition example {Ink} (both [87]) .....	63
Figure 32 - Typical Sketch Example {Pencil} ([37]) .....	64
Figure 33 - Uncertainty functions.....	67

Figure 34 - Edge length distributions: a) Deer0-Deer2; b) Deer3-Deer5 .....	68
Figure 35 - Edge length distributions, accumulative percentages .....	69
Figure 36 - Angle Distribution: a) Absolute Edges; b) Percentage of Total Edges .....	69
Figure 37 - Combination of 100% MinLength and 100% MaxAngle .....	70
Figure 38 - Unconnected Triangles Graph.....	71
Figure 39 - Silhouette changes under rotation.....	73
Figure 40 - RP Sketches: a) Camera; b) Salesman; c) Chess Piece; d) Diving Board .....	75
Figure 41 - Transforming Unit Vector to Desired Edge .....	76
Figure 42 - Different longitudinal Stretch-factors for a Sketchy Line .....	77
Figure 43 - Several Stroke textures .....	78
Figure 44 - Edge extensions: a) static solution; b) view-dependent solution .....	79
Figure 45 - The three possible edge configurations: a) silhouette edge; b) front facing; c) back facing....	80
Figure 46 - Fading function for edges .....	81
Figure 47 - Charcoal sketches: a) Salesman; b) Chess Piece; c) Camera; d) Walker .....	81
Figure 48 - Various hatching suggestions ([24]) .....	82
Figure 49 - Projective Texturing .....	84
Figure 50 - Per-vertex Transparency Shading.....	84
Figure 51 - Hatching-shading using Multi-texturing .....	86
Figure 52 - Using a 3D texture for sketch shading.....	87
Figure 53 - Cross-Hatch Shading: a) Skeleton; b) Statue of Liberty; c) Mouse; d) Dog .....	88
Figure 54 - Performance Results for Hatching.....	88
Figure 55 - Sketch Renderers Performance: a) Total Edges; b) Rendered Edges.....	89
Figure 56 - Rendered Edges vs. Total Edges .....	90
Figure 57 - a) Original Image; b) Video filter; c) Commercial Filter .....	96
Figure 58 - Minimising the capture area.....	98
Figure 59 - Naïve implementation showing background artefacts .....	98
Figure 60 - Painterly Rendering using the Stencil Buffer .....	99
Figure 61 - Painted 3D objects: a) Mouse in offscreen Buffer; b) Mouse in Screen Buffer with Paper Background; c) Dog.....	99
Figure 62 - From Brush Shape to Brush Stroke .....	102
Figure 63 - Oil Painting using Textured Brushes .....	102
Figure 64 - Definitions: Visible Brushes, Object-Pixels, Background-Pixels, and Capture Area .....	104
Figure 65 - Performance vs. Number of Brushes .....	107
Figure 66 - Extremely High Speed .....	110
Figure 67 - Respawnng Behaviour: a) Synchronised; b) Randomly desynchronised .....	111
Figure 68 - Landscape with various Brush Orientations: a) Random; b) Angled; c) Circular.....	112
Figure 69 - Demonstration of multiple effects .....	113
Figure 70 -Brush Variations: a) Original; b) Detail of Spotlight; c) Same Detail with Dot-Brush; d) Dot-Brush; e) Stroke-Brush.....	114
Figure 71 - Video Oil Rendering.....	115
Figure 72 - Image-based rendering, affine transformations .....	121

Figure 73 - Geo-spherical sampling of a real-world object.....	122
Figure 74 - Teapot sampling: a) 15 Degrees; b) 30 Degrees; c) Control-Points and Motion Splines; d) Triangulation based on Control-points.....	123
Figure 75 - Pixel Interpolation through texture mapping.....	124
Figure 76 - Linear interpolation in action .....	125
Figure 77 - Triangulation Problems: a) Initial Triangulation; b) Control-point moves into neighbouring triangle; c) Two possible triangulations of the same control-points .....	126
Figure 78 - Demonstration Set-up .....	128
Figure 79 - Image analysis: a) Edge Detection; b) a possible landmark; c) many possible matches .....	128
Figure 80 - Automatic Landmark identification: a) source; b) target .....	129
Figure 81 - Contour problems: a) proposed landmark; b) possible and incorrect matches .....	130
Figure 82 - Tri-linear sampling for arbitrary rotations.....	131
Figure 83 - CoRgi Object Model (Relevant Part).....	138
Figure 84 - OS Comic Performance .....	140
Figure 85 - OS Sketch Performance (1).....	141
Figure 86 - OS Sketch Performance (2).....	141
Figure 87 - OS Painting Performance.....	142
Figure 88 - Form factor geometry for two patches (after Goral [27]).....	176
Figure 89 - The four mechanisms of light illumination (after [100]).....	178
Figure 90 - Using a virtual environment to account for indirect diffuse illumination.....	179
Figure 91 - An example mapping process.....	181
Figure 92 - Texture Mapping Spaces (after [101]).....	182
Figure 93 - Unfolding of a pyramid.....	183
Figure 94 - Intermediate Surface Mapping Options: a) Reflected Ray; b) Object Normal; c) Centroid Projection; d) Intermediate surface normal (after [101]).....	184
Figure 95 - Bump Mapping: a) Original Geometry; b) The height map; c) Length-Adjusted Normal vectors; d) Bump-perturbed Normals (after [101]).....	186
Figure 96 - Example of Bump Mapping: a) Texture mapped Sphere; b) Same sphere with Bump map (the bump map is a shifted grey map of the texture map) .....	187
Figure 97 - Environment Mapping: a) Object inside an environment; b) Quadrilateral View-frustum (both after [101]) .....	187

## 9.4 Table of Listings

Listing 1 - Standard Render Loop .....	24
Listing 2 - Custom Clear Displaylist .....	25
Listing 3 - Standard Comic algorithm .....	38
Listing 4 - Extended Comic style Algorithm .....	53
Listing 5 - Generic Sketching Algorithm.....	66
Listing 6 - Random Perturbation Sketch Algorithm .....	74
Listing 7 - Object-Segmentation Algorithm.....	77
Listing 8 - Algorithm for Coal Sketching .....	82
Listing 9 - Using projective texturing.....	84
Listing 10 - Generic Hatch-Shading algorithm .....	87
Listing 11 - General Painterly Algorithm .....	95
Listing 12 - Generic Convolution Algorithm .....	96
Listing 13 - Painterly Rendering using Textured Brushes .....	105

## 9.5 List of Tables

Table 1 - System Set-up: Hardware and Software.....	28
Table 2 - Test Set-up: Positions and Orientations .....	28
Table 3 - Universally relevant Object Statistics .....	29
Table 4 - Visual Comparison of Objects.....	31
Table 5- Duo Shading Colour values for a given sample set .....	36
Table 6 - Visual Quality vs. Face Orientation Methods (1) .....	47
Table 7 - Visual Quality vs. Face Orientation Methods (2) .....	48
Table 8 - Performance vs. Face Orientation Method (1).....	49
Table 9 - Performance vs. Face Orientation Method (2).....	50
Table 10 - Comparison of results for various Shininess values.....	55
Table 11 - Performance with Silhouette Colour and Multi-texturing in FPS.....	57
Table 12 - Performance Comparison of Default Renderer vs. Standard Comic and Extended Comic .....	59
Table 13 - Comparison of Face-Sorting vs. Displaylist caching (Extended Comic renderer).....	61
Table 14 - Visual Comparison of Edge-reduction approaches .....	70
Table 15 – First Approximation of Painterly Factors .....	93
Table 16 - Pros and Cons of Background preservation Techniques .....	100
Table 17 - Object Resolution vs. Number of Brushes .....	106
Table 18 - Effects of Object Speed on Painting Behaviour.....	109
Table 19 - Visual Comparison of Painterly Approaches.....	116
Table 20 – Comparative Results for Super-realistic rendering.....	134
Table 21 - Configuration for multiple OS Test .....	140



## 9.6 Sources of Models

Model Name	Original Name	Author	Source/Contact
Camera	Camera	Unknown	www.3dup.com
Cow	Cow	Unknown	www.3dup.com
Hand	Hand	Unknown	www.3dup.com
Skeleton	Skeleton	Unknown	www.3dup.com
Comic Cat	Billy the Cat	D. Proctor	dproc@ozemail.com.au
Comic Ant	CuteAnt	Hou Soon Ming	ming@its-ming.com
Deer	Deer	Hou Soon Ming	ming@its-ming.com
Horse	Horse	Baltsiy Yuriy	www.meta3d.com
Sheriff	Sheriff	Baltsiy Yuriy	www.meta3d.com
Statue of Liberty	LibertyStatue	Mr. Voodoo	www.meta3d.com
Mouse	Mickey	Unknown	www.meta3d.com
Wrench	Wrench	Unknown	www.zygote.com
Chess Piece	Chess	Renzo Del Fabbro	renzo@sunrise.it
Racer	RC_Car	Niels 't Hart	www.bart.nl/~niels
Plane	Biplane	Ian D. West	WEST10397@aol.com

## 10 Appendix A - Photorealistic Techniques

### 10.1 Photorealistic Techniques

Even though this thesis deals primarily with the algorithms, implementation details and performance issues of NPR techniques, we feel that those topics need to be contrasted against the equivalent photorealistic (PR) techniques. This is especially so, since the main drive of the computer graphics community is towards more and faster realism. Only if the effort and energy is understood that goes into the generation of standard, realistic images can we fully appreciate the tricks and subtleties that have been developed to make real-time NPR rendering a possibility. Not only does the NPR renderer live in the somewhat literal shadow of the PR renderer, but it also strives off it. Without the advancements in technology and especially standardisation (of hardware and programming APIs) that have been made to further PR rendering it would be near impossible to implement NPR systems coming even close to real-time. With this in mind we will spend this Chapter discussing the details of the most common and advanced photorealistic rendering techniques.

#### 10.1.1 Ray-tracing ([103])

Ray-tracing is a technique that can be adequately used to model specular interaction (i.e. shiny and perfectly smooth surfaces which reflect and/or transmit an incident ray in exactly one direction). Even though any kind of scene can be produced with this method, it is noticeable that all of them will contain a variety of very reflective objects, preferably spheres or mirrors and at least as many transparent objects imitating any kind of (pure) glass. The downside of ray-tracing is its inability to deal with diffuse interaction (the number of rays having to be spawned at each level would be far too great for any feasible practical implementation) and its general high computational cost. Its advantage is that it provides a number of solutions to the global illumination problem:

- Hidden surface removal
- shading due to direct (local) illumination
- shading due to indirect (global) illumination (i.e. reflection and refraction)
- shadow computation

Ray-tracing works entirely in object space. To minimise the number of rays having to be computed, they are traced backwards and are thus emitted from the viewer's eye through a virtual screen positioned in front of the viewer. The targeted screen resolution is then used to determine the number of initial- or view-rays spawned (naïve ray-tracers might send one ray through the middle of each pixel, but this will result in aliasing artefacts). If a ray strikes the surface of an object (the search for this intersection in a multitude of objects, each in turn with possibly non-trivial surfaces, is the most expensive operation of ray-tracing and can be optimised) the colour for this point is calculated by the superposition of three components:

- the local surface colour (including ambient effect)
- the contribution of any light reflected onto this point (reflection)
- the contribution of any light refracted onto this point (transmission)

The first component is dependent only on the objects properties, independent of its position and orientation (absolute and relative to the viewer), while the other components depend on the object's reflective and refractive properties, as well as its spatial arrangement. To determine the contributions for transmitted and reflected light, rays are spawned from the point of intersection with the first ray. It becomes obvious that this method is naturally implemented in a recursive manner. The depth to which this recursion takes place is another parameter in the ray-tracing process that can be optimised. Since the complexity of the computation grows exponentially with depth, the contribution that is made by a ray spawned at each depth has to be estimated and the rays terminated adaptively. Terminating after depth 1 implements an inherent hidden surface removal for opaque objects.

Shadow computation is done by spawning additional rays (called *shadow feelers*) from an intersection to each of the light-sources. If a totally opaque object lies in between the light source and the surface, then the local intensity contribution is reduced to the ambient value. If the blocking object is partially transparent, then the local intensity term is attenuated accordingly. Including refraction in the calculation of the shadow-feeler rays cannot easily be achieved without major modifications to the main algorithm. In addition to this the number of light sources is limited in practice when computing shadows. It is found in practice that *main* rays need only be traced to an average depth between 1 and 2 [29] before contributions from reflected or refracted rays become negligible. But to compute accurate shadows,  $n$  shadow-rays have to be spawned for each intersection, if  $n$  is the number of light sources used. This implies that the cost for the computation of shadows quickly dominates the total ray-tracing cost.

## Optimisations

### *Bounding volumes*

Determining the intersections between objects and rays is a non-trivial task for objects of arbitrary geometry (the reason why ray-traced scenes preferably show spheres - for which the intersection test is trivial). Bounding volumes of simpler geometry help the optimisation process and hierarchical structures of bounding volumes can be used for clustered objects. Nonetheless, setting up the scene including the bounding volumes then becomes a non-trivial task and is not easily automated. An intersection test for arbitrary planar polygons is as follows:

- obtaining an equation for the plane containing the polygon
- checking for intersection with this plane
- checking if the intersection is contained within the polygon

The last step listed here is trivial but may be computationally expensive as it depends on the number of vertices that make up the polygon.

### *Space partitioning*

Another approach is to divide object space into regions containing objects and those that do not. Initially the object space is restricted to a cube-shaped region and marked either empty or non-empty. If it is marked non-empty the (recursive) process is taken one level further and the initial cube is divided into 8 sub-cubes of equal size. The process is then repeated for each of the sub-cubes until they are either empty or contain only one object. Determining whether an object lies within a given cube is trivially achieved by testing each of the vertices to lie within the range dictated by the cube. Any given ray now only has to be tested for intersection if it passes through a non-empty cube.

### *Distribution*

Due to its recursive nature the ray-tracing algorithm can easily be distributed in a parallel computing environment.

### *Anti-aliasing*

Since a naïve ray-tracer will produce aliasing. To combat this, a non-uniform over-sampling approach is suggested by Mitchell [55], which works in three steps:

- A ray tracing solution is constructed at a low sampling density (e.g. one ray per pixel as in the naïve ray-tracer)
- This solution is then used to determine areas that require super-sampling
- An image is constructed from these non-uniform samples by an appropriate reconstruction filter

### **10.1.2 Radiosity ([103])**

The Radiosity method of light interaction was developed at Cornell University by Goral et al [27] as an answer to the limited capabilities of the ray-tracing method (see Section 10.1.1) which can only deal with specular light-interaction thus creating images with a *shiny-plastic* look to them. The radiosity method divides all surfaces up in (largish - otherwise the computational complexity exceeds most practical limits) patches that are assumed to exhibit constant physical properties. This limited amount of spatial resolution makes it practically impossible to model specular interaction. The radiosity and ray-tracing method can thus be considered mutually exclusive and complementary to each other. Attempts to unify both methods have been made.

The radiosity method is based on conservation of energy, which places the rather artificial constraint on rendered scenes that they have to be inside a closed room through which no energy can escape. Energy is input to the system through emitting surface patches. This allows any patch to act as a light emitter and makes it possible to create light sources of virtually any conceivable shape.

Several natural phenomena that were previously not incorporated in any lighting model could now be addressed:

- *colour bleeding* from one surface to another,
- shading with shadow envelopes, and
- penumbræ along shadow boundaries

Since the radiosity method is an object-space algorithm which is concerned with screen-space projection only in its last processing step, its main solutions can be re-used to create views from any possible position and orientation. This means that a lot of effort goes into arriving at the above-mentioned solution, but once this has been done, re-rendering of the scene from different viewpoints becomes trivial. The obvious assumption is that objects within the scenes are not animated.

Another assumption made is that all surfaces within the scene are perfect diffusers (or ideal Lambertian surfaces). In general, radiosity  $B$ , is defined as 'the energy per unit area leaving a surface patch per unit time' and is the sum of the emitted and the reflected energy:

$$B_i dA_i = E_i dA_i + R_i \int_j B_j F_{ij} dA_j$$

where the reflected energy is arrived at by multiplying the reflectivity of patch  $i$  by the fraction of energy of patch  $j$  that arrives at patch  $i$ . This fraction  $F_{ij}$  is called the *form factor* and its calculation is major part of the radiosity computation. Siegel and Howell [85] state the following reciprocity relationship:

$$F_{ij} A_i = F_{ji} A_j$$

so that division by  $dA_i$  gives:

$$B_i = E_i + R_i \int_j B_j F_{ij}$$

By using a discrete environment with finite size patches while assuming constant radiosity for each patch we can write the integral as a sum and arrive at:

$$B_i = E_i + R_i \sum_{j=1}^n B_j F_{ij}$$

where  $n$  is the number of discrete patches in the environment. Since an equation exists for each of the  $n$  patches there exists a set of  $n$  simultaneous equations that can be written in Matrix notation:

$$\begin{bmatrix} 1-R_1F_{11} & -R_1F_{12} & \dots & -R_1F_{1n} \\ -R_2F_{21} & 1-R_2F_{22} & \dots & -R_2F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -R_nF_{n1} & -R_nF_{n2} & \dots & 1-R_nF_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

In the above matrix the  $E_i$  are only non-zero for the patches that act as light emitters thus providing illumination and can be considered the input to the system. The  $R_i$  are either known from material properties or can be calculated. The  $F_{ij}$  are dependent on the geometry of the environment and need to be calculated. Some simplifications can be made: For any plane or convex surface  $F_{ii}=0$  since none of the radiosity leaving the surface will be able to strike itself. From the definition of the form factor it is evident that the sum of any row of form factors is unity.



As was mentioned above the form factors are a function of the geometry only (and not view-dependent) so that the solution obtained above produces a single value for each of the surface patches, the information of which can be inserted into a standard Gouraud renderer to give an interpolated solution across all patches.

The form factor is calculated by taking into consideration the relative orientation between two patches and their distance from each other. In physical terms it is describing the radiative energy leaving surface  $A_i$  that strikes  $A_j$  directly, divided by the radiative energy leaving surface  $A_i$  in all directions in the hemisphere surrounding  $A_i$ :

$$F_{A_i A_j} = F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos \phi_i \cos \phi_j}{\pi r^2} dA_j dA_i$$

the graphical explanation of which can be found in Figure 88. In most practical situations the two patches may be totally or partly occluded from each other so that an occlusion factor has to be included into the calculation. Except for a few specific cases the above-mentioned double integral is difficult to solve.

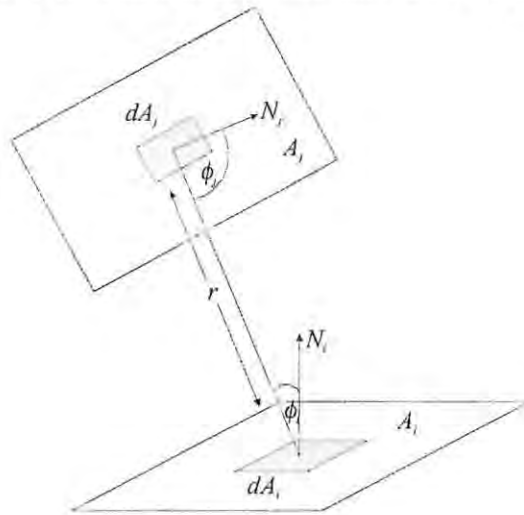


Figure 88 - Form factor geometry for two patches (after Goral [27])

For this reason Cohen and Greenberg [11] developed a numerical method for evaluating the form factors, which approximates the hemisphere by a hemi-cube of finite resolution (the units of spatial resolution being called pixels which are not related to the conventional use of the word). In their approach every patch in the environment is projected onto pixels comprising the hemi-cube. If two patches are projected onto the same pixel, then the further one is ignored (similar to a z-buffer, except that at this stage no intensities are depth-sorted, but instead the visibility of patches is determined for the calculation of the form factors). With this approach several other simplifications can be made (e.g. use of look-up tables for standard form-factors on areas of the hemi-cube) that allow for a significant speed-up. The modified radiosity algorithm then comprises the following steps:

- Computation of the form factors,  $F_{ij}$
- Solving the radiosity matrix equation
- rendering by injecting the results of stage 2. into a bilinear interpolation scheme
- repeating stages 2. and 3. for each of the colour bands of interest

Since the calculation of the form factors depend solely on the geometry of the environment, their values can be used for different views (as mentioned above) but also for different values of  $R_i$  or  $E_i$ . This means that the material properties of the surfaces can be modified without affecting the form factors as well as light sources be switched on or off at will. All that needs to be done is solving the matrix equation of stage 2 again. For different view-perspectives only stages 3 and 4 have to be repeated.

An upper limit on the practical complexity and accuracy of the scene is determined by the processing time allowed and the storage resources available. For example the time to compute the form factors is of order  $O(n^2)$  in the number of patches. In turn each patch has a hemi-cube of a certain resolution associated with it. This resolution impacts on the computational complexity as well but is responsible for the projected accuracy of the rendering (the higher the resolution, the better the image accuracy). Both the number of patches and the resolution of hemi-cubes are limited by the memory available to the system.

## Optimisations

### *Explicit shadow calculations*

One optimisation by Nishita and Nakamae [58] predicts the umbrae and penumbrae of light shadows explicitly with shadow volume algorithms and use this information to adjust the surface division into patches accordingly to that regions of high radiosity gradient are subdivided to a higher resolution than other regions. With this approach they can concentrate the computing resources onto areas of finer visual detail.

### *Progressive refinement*

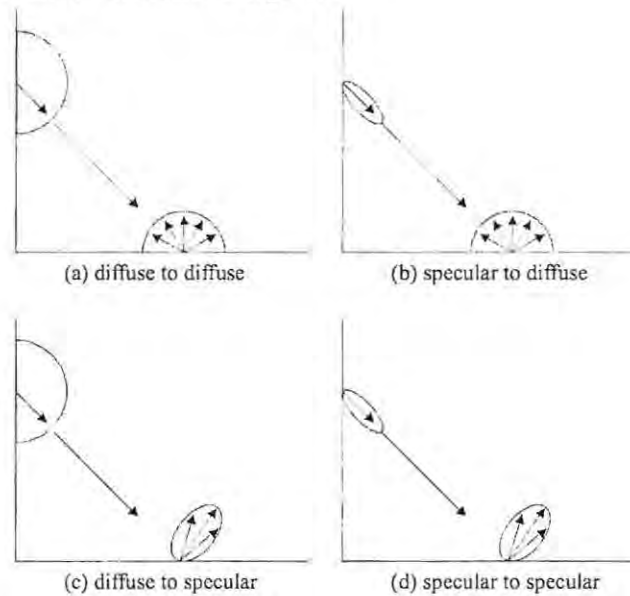
This technique by Cohen et al [12] uses a first coarse radiosity solution to identify areas of large gradient which may need further refinement. In successive re-iterations the desired degree of accuracy is achieved. This approach makes use of the fact that as far as the radiosity solution is concerned, the cumulative effect of elements of a subdivided patch is identical to that of the undivided patch. In other words, the amount of light reflected by a given patch is not affected by it being super-sampled into several patches. This enables the calculations to be performed as if only the patch under consideration is sub-divided, while all other patches remain the same, which greatly reduces the number of patches to be considered. In addition to this previously obtained patch-values can be re-used in the refinement process even if the patches have been sub-divided in the process.

### **10.1.3 Hybrid (Ray-tracing and Radiosity [103])**

As was already mentioned in the discussions of Ray-tracing and Radiosity (Sections 10.1.1 and 10.1.2 respectively), these two methods are mutually exclusive and cater for only one type of either specular or diffuse light-interaction. Any system claiming to realistically model light-interaction would have to incorporate both. Some attempts in this direction have been made, but the computational costs involved are usually prohibitive.

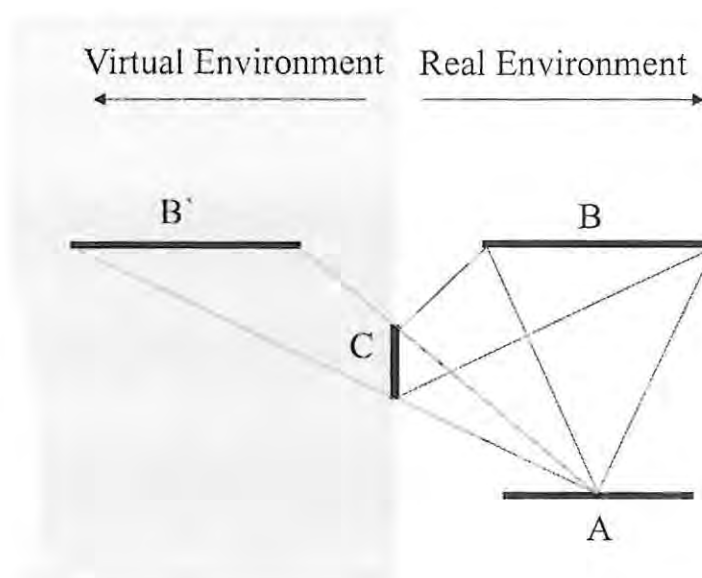
Immel et al. [37] first extended the standard radiosity method to include specular interaction by incorporating a bi-directional reflectivity function into the basic radiosity equation and adopting a view-independent solution for the specular interaction. While the diffuse illumination changes rather slowly over a surface, this cannot be said about specular illumination. For this reason the surface-subdivision necessary to model the specular component correctly results in massive computational overheads for anything but simple scenes.

A two-pass approach was later developed by Wallace et al [102], of which the first pass (named the preprocess) consists of an enhanced radiosity solution. This approach is based on Wallace's notion of four mechanisms of illumination, as illustrated in Figure 89.



**Figure 89 - The four mechanisms of light illumination (after [102])**

The mechanism labeled (a) is handled in the standard radiosity equation. Mechanism (d) is accounted for in the ray-tracing method. Modeling mechanisms (b) and (c) is the achievement of Wallace's method. The aforementioned enhancement in the preprocess is the inclusion of diffuse transmission (translucency) and specular-to-diffuse transport, the latter of which happens when a diffuse surface patch *sees* another similar patch via the specular reflection of a shiny surface. The hemi-cube method is still used to determine the form-factors, but another (back-facing) hemi-cube is used in order to determine form-factors for translucency. The diffuse intensity component due to mechanisms (c) {direct} and (d)& (b) {indirect} are thus taken into consideration, resulting in a view-independent solution for these mechanisms. The only specular surfaces considered are perfect mirrors and the additional possible paths are taken into account by even more form-factors, this time derived from a virtual environment, as in Figure 90.



**Figure 90 - Using a virtual environment to account for indirect diffuse illumination**

In Figure 90 we see the direct diffuse illumination of patch B onto A as green lines, but if there happens to be a specular surface C, then the additional indirect diffuse illumination of A due to B can be modelled by creating a virtual (mirrored) environment with a virtual patch B', with the same properties as patch B. These virtual environments are seen as the main disadvantage of the method, since the form-factor calculations are mainly dependent on the complexity of the environment. Augmenting the environment with additional (virtual) environments is therefore a heavy burden on the computational complexity.

The second stage of this approach is called the postprocessor and deals with the specular illumination of the scene. Here, a view-dependent solution is calculated from the results of the pre-process. The fact that the solution is view-dependent means that a ray-tracing approach can be utilised. The ray-tracer used by Wallace is enhanced in that it uses a projection frustum rather than a naive infinitely thin ray of light (similar to Figure 90). Incoming diffuse intensities contributing to each reflection frustum are calculated by linear interpolation from the preprocess patch vertex intensities. The frustum is implemented as a pyramid with square base of  $n$  by  $n$  'pixels', where  $n$  is adaptively changed as a function of recursion depth. The incoming intensities that stream through each pixel are weighted and summed to model the specular spread and shape of the specular interaction.

#### **10.1.4 Local Illumination Model**

It becomes evident from the above discussion that rendering techniques which take into account reflections and refractions of diffuse and specular light components are extremely computationally expensive and are not realisable in real-time on common hardware platforms. As a compromise (and a further extreme simplification) the local illumination model (LIM) only takes into account the direct interaction between a light source and an object. All other objects are disregarded in the shading computations. This means that light cannot be reflected off objects, it cannot refract through objects and objects cannot cast shadows on other objects (in fact not even themselves). To clarify the matter, we are not saying that these effects cannot be produced (in fact the following Sections deal with creating such effects), but that they are not part of the LIM. Several graphics APIs (most commonly used are OpenGL

and Direct3D of the DirectX set of APIs) implement the LIM and the popularity of 3D games and other 3D applications have led to a well defined standardisation of the LIM on modern graphics accelerators. Independent of the implementation details (for details on engine architecture, see [83] for OpenGL and [54] for Direct3D) the following factors are common to most APIs and represent the key to speedy rendering:

- The number of lights is limited (usually eight or fewer)
- Shading information is only computed at vertices. Filling is performed by interpolating colour information at vertices (this implies that for a flat surface to be properly lit, it has to be relatively finely tessellated)
- Material properties are limited to ambient, diffuse and specular reflection coefficients, shininess factor and emission value. These model surfaces such as plastic or metal relatively well

While this set of functionality allows thousand of primitives to be rendered with basic smooth shading and in real-time the objects rendered in such a manner have a distinct plastic and computer-generated quality to them. The methods and techniques that can be applied to emulate more realistic surfaces and light interaction are discussed next. In general these can be considered coarse approximations intended to please the eye rather than simulate nature in a realistic manner.

## 10.2 Realism Enhancing Techniques

A variety of techniques have been developed to increase the realism of images produced by the LIM, while retaining as much performance as possible. Most of these techniques fall into the category of mapping techniques, which are discussed next.

### 10.2.1 Mapping Techniques

Mapping in the most general sense refers to some function  $f$  that maps values from some input space  $I$  to some output space  $O$ . The dimensionality of the input and output spaces can differ.

$$f(i_0, i_1, \dots, i_m) = (o_0, o_1, \dots, o_n)$$

Equation 3- General Mapping

The generality of Equation 3 has two implications in a graphical context: The formula can be applied to a large amount of problems and the implementation can be well standardised. One of these standardisations is that we can usually think of the input values as contained inside a two dimensional image (whether the pixels of the image represent, colour, alpha or vectors is not important at this point). Examples of more dimensional input exist (e.g. 3D textures), but are rarely used.



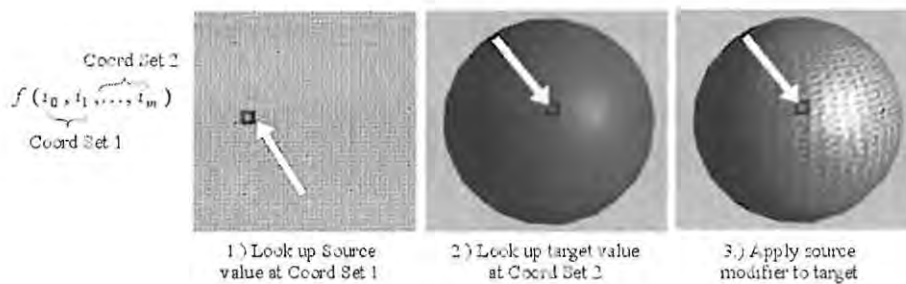


Figure 91 - An example mapping process

Figure 91 shows a typical mapping process. The general mapping is broken down into sub-parts: A set of input co-ordinates specifies the source and target locations. A modifier from the source location is processed and applied to the target location. With this process targets are usually altered, rather than overwritten, meaning that some degree of feedback takes place. While the example shows texture mapping, the same process can be applied to bump mapping and a number of other mapping techniques, which are explained below.

#### 10.2.1.1 Texture Mapping ([103])

This Section explains the workings of texture mapping. As some people actually consider texture mapping a kind of super-class to other mappings such as environment or bump mapping, we go into some detail to show common characteristics and unique features of texture mapping.

Texture mapping is the process of transforming a texture onto the surface of a three-dimensional object. Texture maps were originally employed to give three-dimensional objects a more complex and interesting appearance. Up to that point objects would exhibit a distinct plastic look, due to the Phong reflection model generally employed and objects could mainly be differentiated by colour and shape. The introduction of texture mapping allows objects to appear with different surface properties. These properties are not limited to colour modulation, but may give the impression of surface modulation as well (as for example a "bark of a tree" texture might suggest).

According to Watt [103] there are three major considerations in texture mapping:

**1. What attribute or parameter of the model or object is to be modulated to produce the desired textural effect?**

The following is Watt's modified version of the original classification by Heckbert [31]:

- **Surface colour** (diffuse reflection coefficient(s))
- **Specular and diffuse reflection** (environment mapping), first developed by Blinn and Newell [7]. Considered a class of its own as opposed to a sub-class of texture mapping. One reason given for this is that all texture mapping techniques fix a texture onto an object independent of

that object's position or orientation. Environment mapping on the other hand takes both these properties explicitly into consideration.

- **Normal Vector** perturbation (Bump mapping)
- **Specularity**. This was again invented by Blinn [8] but is not used often. The quantity modulated is the surface roughness in the Cook-Torrance reflection model and could be used to generate effects such as textured paints.
- **Transparency**: Examples of this technique are given by Gardener [21]. Applications include creation of clouds or chemically etched glass.

## 2. How is the texture mapping to be carried out?

Given that a texture is defined in a texture domain and an object exists as world space data, we need to define a mapping between these domains.

According to Watt there are several possible texture domains, i.e. one-dimensional, two-dimensional or three-dimensional (see figure below). Even though the two-dimensional domain is the most popular, the other domains are claimed to be considerably easier to handle even though they are somehow restricted in the textural effects producible.

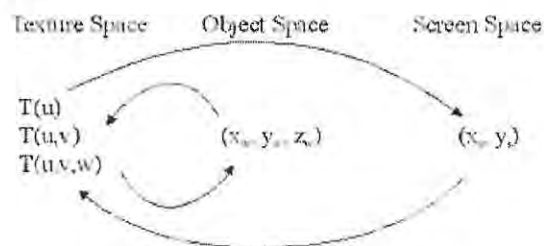


Figure 92 - Texture Mapping Spaces (after [103])

The two major implementations used are:

1. mapping from texture space to object space (usually a 2D to 3D transformation) and then performing the viewing projection to screen space.
2. mapping directly from texture to screen space (usually a 2D to 2D transformation).

In the latter case, the screen space is usually uniformly sampled and the inverse or pre-image of a pixel in texture space is formed. The area thus produced in the texture domain is then sampled and filtered and a texture value then returned to the pixel. Filtering methods are easier to implement for inverse mapping.

Difficulties arise in the general case, where the transformation from screen space to texture space is non-linear resulting in a pixel pre-image that is a 'curvilinear quadrilateral' whose shape varies as the pixel position changes.

Texture mapping requires special anti-aliasing treatment because it tends to produce worse aliasing artefacts than other techniques associated with image synthesis.

### Mapping onto polygon mesh models

In the general case, where three dimensional objects are comprised of a number of polygons which are to be texture-mapped we need to find a way to parameterise the interior of these polygons in order to apply the texture onto the polygons which are only defined by their vertices. This may result in distortions and/or discontinuities, which, depending on the geometry of the object, may have to be dealt with.

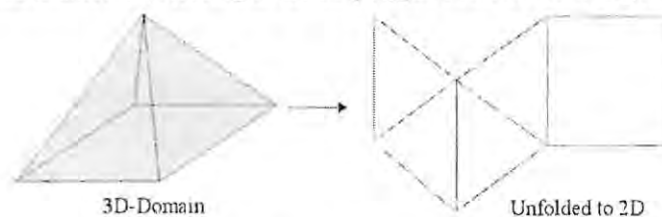
Heckbert [31] fully describes parameterisation techniques and points out that a triangle is the easiest polygon to parameterise. In addition to this it is always guaranteed that all vertices of a triangle lie in exactly one plane so that for any point  $(x,y,z)$  on the object there exists a linear relationship with the texture-co-ordinates  $u$  and  $v$ :

$$(x,y,z) = (A_x u + B_x v + C_x, A_y u + B_y v + C_y, A_z u + B_z v + C_z)$$

For polygons with more than three vertices this parameterisation is not adequate as topological problems may arise. How to deal with these problems is described next:

### Unfolding the polygon mesh

A technique developed by Samek et al. [79] shows us how to map the surfaces of an object onto a single plane, by unfolding adjacent faces. For this pivoting of each polygon in the object about the edge with its neighbour to work properly, the object has to exhibit a single surface topology. Projection of the texture space onto the unfolded object is said to be easily achieved so that an effective mapping of object vertices into  $(u,v)$  texture-co-ordinates takes place. An example of this can be seen in Figure 93, where a three dimensional pyramid is unfolded, resulting in the 2D polygon structure on the right.



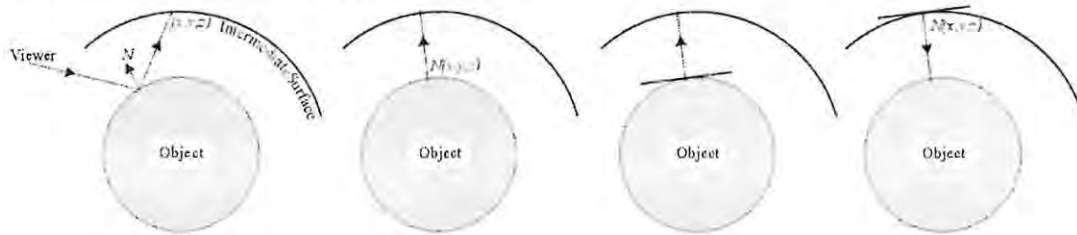
**Figure 93 - Unfolding of a pyramid**

Already from this example it should become evident, that discontinuities may occur during the mapping process at certain edges of the unfolded object.

To make this technique work, adjacency information is stored in a special data-structure. The angle between polygon  $n$  (current) and polygon  $n-1$  (last) is established and incorporated into a rotation that pivots polygon  $n$  around the common edge between itself and polygon  $n-1$ , resulting in both polygons now lying in the unfold plane.

### Two-part mapping

Another method restricts the object's topology less and is dependent only on the object's geometry as opposed to its parameterisation, thus overcoming the above-mentioned difficulties. Introduced by Bier and Sloan [6] this method uses an intermediate shape to initially project the texture on, similar to environment mapping (for which it can also be used)



**Figure 94 - Intermediate Surface Mapping Options: a) Reflected Ray; b) Object Normal; c) Centroid Projection; d) Intermediate surface normal (after [103])**

The mapping process is divided into two steps:

1. Mapping from 2D texture space onto the intermediate 3D surface, which may be any adequate shape of simple geometry (e.g. sphere, box, cylinder) for which a straight-forward mapping exists. This is called S mapping:

$$T(u, v) \rightarrow T'(x', y', z')$$

2. Mapping of the intermediate surface onto the actual object's surface. This is referred to as O-mapping

$$T'(x', y', z') \rightarrow O(x, y, z)$$

As the geometry of the intermediate object should be chosen so that a well-known and inexpensive parameterisation exists, the first step can be considered trivial. The second step can be performed in one of several possible ways, as illustrated in Figure 94. The colour for a given point on the surface of the object  $O(x, y, z)$  can be derived from:

- the intersection of the reflected view ray with the intermediate surface
- the intersection of the surface normal at  $(x, y, z)$  with  $T'$
- the intersection of a line through  $(x, y, z)$  and the object's centroid with  $T'$
- the intersection of the line from  $(x, y, z)$  to  $T'$  whose orientation is given by the surface normal at  $(x', y', z')$

Bier and Sloan identify three classes of S-mapping and the listed four classes of O-mapping giving a total of 12 possible combinations. They also mention that not all of these combinations are useful, while they given special names to others for the special purpose that they fulfil.

### 10.2.1.2 Bump Mapping ([103])

Developed by Blinn [8] this method enables a surface to appear as exhibiting a macroscopic relief-structure without the need to model this relief explicitly in geometry. Instead the effect is achieved by perturbing the surface normal angularly according to information stored in a 2D bump map or procedurally. Since in the local reflection model light-intensity is mainly a function of the surface normal, these variations of the normal vector trick the viewer into believing to see more detail than is actually existent. This can become apparent, when bump-mapped objects intersect or at object-edges; in this case the cross-section produced will follow the geometric information of the object without taking into account the "virtual" bumps of the bump-map, creating an unrealistic impression.

Since the normal-vector perturbation must be independent of the position and orientation of the surface (otherwise the bump-mapping would be inconsistent through animations), it must be based on the local surface derivatives.

If we let  $O(u,v)$  be a parameterised function, representing the position vectors of points  $O$  on the surface, then the normal to the surface at any given point is given by:

$$N = O_u \times O_v$$

where  $O_u$  and  $O_v$  are the partial derivatives of the surface at point  $O$  lying in the tangential plane. Two other vectors that lie in the tangential plane are:

$$A = N \times O_v \text{ and } B = N \times O_u$$

If  $D$  is the perturbation vector, derived from components of these two vectors, then the perturbed vector  $N'$  is

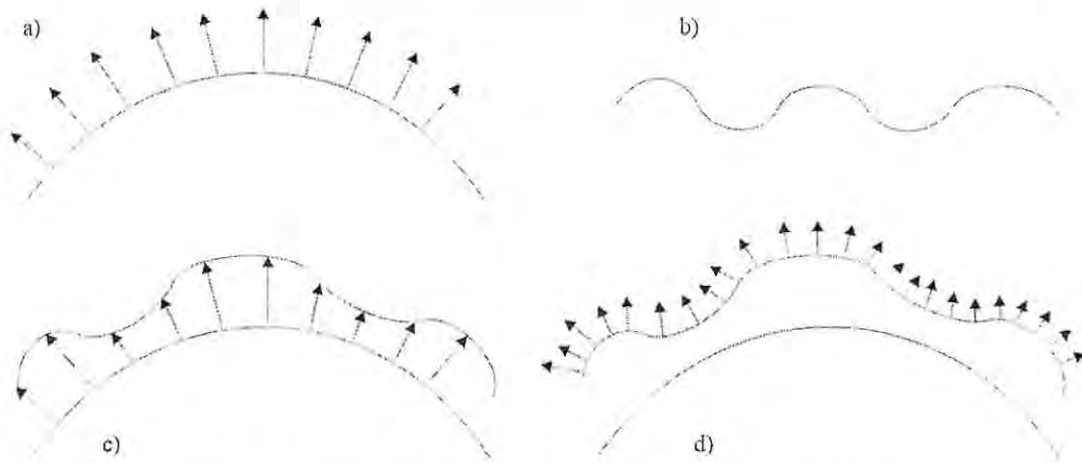
$$N' = N + D$$

In the coordinate system that is formed by  $A$ ,  $B$  and  $N$  the vector  $D$  is independent of the global orientation and position of the surface, which is what we wanted to achieve. Now we need to determine the value of  $D$ .

$$D = B_u A - B_v B$$

where  $B_u$  and  $B_v$  are the partial derivatives of the bump map  $B(u,v)$ . This means that we define a bump map as a height-field (or displacement function) and use its derivatives at the points  $(u,v)$  to determine  $D$ .





**Figure 95 - Bump Mapping: a) Original Geometry; b) The height map;  
 c) Length-Adjusted Normal vectors; d) Bump-perturbed Normals (after [103])**

Blinn has identified several problems with this method:

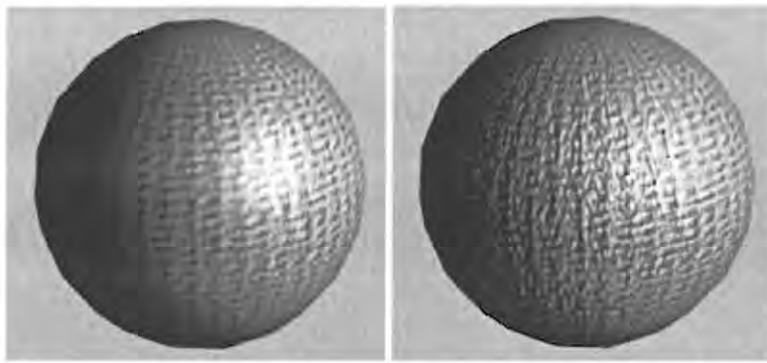
- Dependence on the scale of the object: If the surface definition function is scaled by a factor of two then the length of the normal vector is scaled by a factor of four, while the length of  $D$  is only scaled by two. This leaves the wrinkles smoothed out, which is undesirable. He therefore suggests a scale-invariant version of  $D$ , namely:

$$D' = a \cdot D \cdot \frac{|N|}{|D|}, \text{ where } a = \sqrt{B_u^2 + B_v^2}$$

- Anti-aliasing is another problem addressed by Blinn, which can be attended to by area filtering of the bump map.

Another method to create perturbations is to use the height field of the bump-map to actually distort a fine enough mesh. As this is a rather expensive operation, since all the geometric detail of the perturbed object has to be stored, this method should only be used to deal with the above-mentioned cross-Section artefact.

An extension to this method was created by Kajiya (1985) and is called 'frame mapping'. The difference is that instead of just the normal-vector a so-called frame-bundle is perturbed. A frame-bundle (for a surface) refers to a local co-ordinate system, given by the tangent, binormal and normal to the surface. Kajiya claims that with his approach a mapping of the directionality of surface features such as hair and cloth can be achieved.



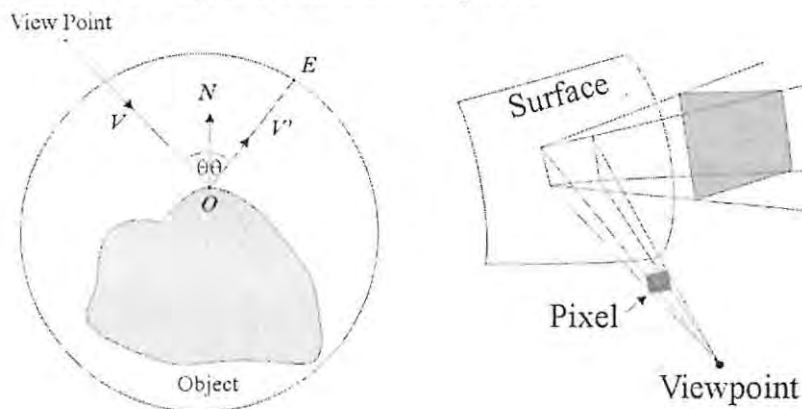
**Figure 96 - Example of Bump Mapping: a) Texture mapped Sphere;  
b) Same sphere with Bump map (the bump map is a shifted grey map of the texture map)**

Figure 96 shows several aspects of bump mapping very well: Firstly, the previously very smooth sphere of Figure 96a) attains a very convincing macro-structure and secondly, this structure is not part of the geometry itself. This is very noticeable at the perfectly straight silhouette edges and due to the fact that not the geometry of the object is perturbed, but the normals of the surface.

Even though Blinn's method can be used with great effect, it is not supported (yet) by most hardware (Blinn's method requires per-pixel lighting calculations, whereas most hardware supports only per-vertex lighting). There are some tricks to overcome this problem and some very interesting material is available on the internet (see [20] and [64] for further information).

### 10.2.1.3 Environment Mapping ([103])

Even though this technique is sometimes referred to as a texture mapping technique there is a significant difference. In normal texture mapping the texture projected onto the surface of an object depends solely on the geometry of the object. In addition to this, environment mapping is a function of the position and orientation of the object in the world space and the relation to the viewer (view-vector-dependence). In essence environment mapping can be considered a simplification of ray tracing, where only the reflected ray is traced and the recursive process is terminated at depth two.



**Figure 97 - Environment Mapping: a) Object inside an environment;  
b) Quadrilateral View-frustum (both after [103])**

If  $\theta$  is the angle between the surface normal  $N$  and the view-vector  $V$ , then there exists a reflected vector  $V'$  that can be described by

$$V' = V + 2N \cos \theta \text{ (as in Figure 97a)}$$

If the object is placed inside an environment object (typically a sphere or box), then the reflected ray will intersect with this enclosing shape. The environment object will have one or more texture maps associated with it so that a look-up can take place. The colour-value at point  $E$  where the reflected ray hits the surface of the environment shape determines the colouring of the point of reflection  $O$ . As in practice we deal with finite size pixels, we are faced with four rays emitting from the viewpoint, as in Figure 97b). In order to obtain the correct colour value for the pixel to be painted, we have to average the quadrilateral area spanned and projected onto the environment map by the rays.

Another view of environment mapping is that of extending the limited (in number of light sources) point light source Phong illumination model. Here, the environment map represents an infinite amount (actually depending on the resolution of the environment map) of light sources with the specific geometry of the environment object. For example area light sources like long fluorescent lamps can be simulated by high intensity white values in the environment map.

In order to increase efficiency of this technique some assumptions can be made:

- The reflective object is near the centre of the environment object
- The reflective object is small compared to the environment object

With these assertions, the environment map can be pre-filtered and a direct look-up into a colour-table can occur. In the case of a spherical environment object this would mean indexing a look-up table based on the longitudinal and latitudinal direction of the reflected view-vector. This is a very efficient approach, as the pre-filtering and mapping can be done once off. In some cases the environment information may be re-used to texture-map the actual environment (rooms, horizon, etc.) and thus improve even more on efficiency. It is stated though that for practical reasons, the geometry of a cube is easier computed than that of a sphere of reasonable spatial resolution. For example, to create a realistic horizon map, the images being mapped onto the inside of the cube would have to be pre-distorted to create a spherical impression. The available mappings to achieve this are straightforward, but in many cases introduce discontinuities at the cube edges and corners. The target visual quality will have to be traded off against processor usage.

There are also some disadvantages: The initial assumptions limit the applicability of the process. Even though the environment object can always be made big enough to validate the assumptions (which is implied when using a parameterised environment-map) objects will not be correctly displayed close to environment boundaries. In addition to this, it may be difficult to map certain environment geometries onto a sphere. The alternative of using a box as the environment object has the advantage of naturally modelling geometries like rooms. In this case, building the environment maps can simply be achieved by taking actual photographs of all the walls of the room to be modelled. The disadvantage here is that the look-up procedure becomes somewhat more involved. Miller and Hoffman have devised a system that

uses a cube projection as an intermediate step in producing a latitude-longitude map. In any case it has to be considered that under normal circumstances the environment map will not include any objects inside the environment (including the reflective object) making it impossible to achieve self-reflection or reflection of neighbouring objects. It is conceivable though to recursively add objects into the environment map and thus allow self- as well as mutual reflection.

It is stated that environment mapping may even have some advantages to naïve ray-tracers in that several filtering steps necessary for anti-aliasing may be dealt with once off while creating the environment map. In addition to this some diffuse interaction can be simulated (not possible in generic ray tracing) by using two environment maps, a diffuse and specular respectively. The diffuse map will be a blurred version of the specular one (which is the one discussed above). The blurring can be achieved by convolving the specular map with a Lambert's law cosine function. Indexing this map is done using the surface normal  $N$  instead of the reflected view-vector. Depending on the object's surface properties different ratios of diffuse and specular illumination are then added to the pixel's final colour value. As the diffuse map will not contain any high frequencies, it may be stored at an extremely low resolution.

